Comparison of Operating System Complexity

Dan-Simon Myrland
dansimon@radiotube.org

ABSTRACT

It is plainly obvious that computer operating systems are growing increasingly complex every year, and have been for some time now. In the early days of UNIX a PDP-11 with ¼ Mb of ram and 30 Mb of diskspace served its 100 users well, whereas today a computer with 10,000 times more resources is not even adequate for a single user. The increased complexity does not only tax our hardware but also our minds. Whereas Dennis Ritchie and Ken Thompson at Bell Labs wrote the entire UNIX operating system and userland in a matter of weeks in the late 60's, you would be hard pressed to find a developer today that can finish a company webpage within a year.

Naturally you can *do* a lot more with a computer today then you could in the 70's, but at what cost? This article does not give a definitive answer to the correct balance between providing the necessary features and keeping things simple, instead it simply analyzes the complexity of operating systems, and their various components. Such analysis is illuminating in itself and can provide hints to the above question. Only open source UNIX-like operating systems are analyzed, due to legal requirements and the need for commonalities in order to make comparisons.

Table of Contents

1. Preliminary information

1.1. Abbreviations used

doc	Pages of user documentation	src	Total lines of system source code
man	Manual pages (55 lines/pg)	pkg	3rd party packages in repository
bin	Number of programs	mem	Memory usage at startup
files	Number of files	hdd	Disk space used after installation
conf	System configuration files	K, M	times a thousand, million
pss	Running processes at startup	n/a	Information not available

1.2. Operating Systems used

HISTORIC:			ALTERNATIVE:		
UNIX	V1	1971-11	9ferno	latest	2022-10-07
UNIX	V5	1974-06	Plan9Port	latest	2022-10-07
UNIX	V6	1975-05	9legacy	latest	2022-10-07
UNIX	V7	1979-01	9front	latest	2022-10-07
BSD	4.1	1981-06	Minix	3.4.0rc6	2017-05
UNIX	V8	1985-02	Minoca	latest	2022-10-07
BSD	4.3	1986-06	SerenityOS	latest	2022-10-07
UNIX	V10	1989-10	Haiku	latest	2022-10-07
LINUX:			BSD/SOLARIS:		
Tiny Core	current	2022-10-07	OpenBSD	7.1	2022-04-21
Alpine	3.16.2	2022-08-09	NetBSD	9.3	2022-08-04
Debian	11.5.0	2022-09-10	DragonFly BSD	6.2.2	2022-06-09
Slackware	15.0	2022-02-03	FreeBSD	13.1	2022-05-16
openSUSE	15.4	2022-06-08	OmniOSce	latest	2022-10-07
AlmaLinux	9.0	2022-05-26	OpenIndiana	Hipster	2021-10

1.3. Statistics

os	man	bin	files	conf	pss	src	pkg	mem	hdd
v1	~160	65	329	4	n/a	~21K	n/a	n/a	0.8Mb
v5	~140	94	<528	4	4	56K	n/a	11Kb	~2.0Mb
v6	~175	120	1855	7	4	83K	n/a	11Kb	4.0Mb
v7	264	164	2K	8	5	165K	n/a	19Kb	9.7Mb
41bsd	509	290	2K	20	9	402K	n/a	22Kb	13Mb
v8	501	290	10K	20	6	467K	n/a	11Kb	26Mb
43bsd	1272	351	10K	40	13	654K	n/a	174Kb	20Mb
v10	1513	525	<21K	n/a	n/a	1.8M	n/a	n/a	<157Mb
9ferno	1201	688	14K	<3	21	1.2M	n/a	640Kb	156Mb
p9p	807	268	8K	<3	9	435K	n/a	~30Mb	105Mb
9legacy	1184	816	20K	<5	53	1.8M	~300	~5Mb	592Mb
9front	1213	962	37K	<5	69	1.7M	~300	~15Mb	544Mb
Minix	12K	630	17K	244	46	6.8M	4280	54Mb	911Mb
Minoca	n/a	135	649	44	4	677K	269	7.6Mb	26Mb
Serenity	227	306	15K	19	30	545K	257	110Mb	934Mb
Haiku	21K	613	32K	61	22	2.9M	3597	237Mb	695Mb
OpenBSD	15K	836	25K	446	45	30M	10K	56Mb	1.5Gb
NetBSD	35K	1146	37K	424	24	41M	19K	112Mb	1.1Gb
DragonFly	28K	927	28K	337	125	10M	29K	20Mb	398Mb
FreeBSD	64K	967	19K	603	41	17M	30K	71Mb	1.9Gb
OmniOSce	113K	1303	75K	1009	41	<12M	1801	83Mb	649Mb
OpenIndiana	82K	2356	224K	1917	82	<12M	6984	277Mb	7.5Gb
Tiny Core	n/a	370	25K	86	64	23M	2407	33Mb	29Mb
Alpine	n/a	447	47K	342	69	23M	17K	43Mb	135Mb
Debian	13K	3114	293K	1829	161	n/a	60K	497Mb	3.8Gb
Slackware	72K	6253	681K	2490	99	429M	9180	97Mb	16Gb
openSUSE	27K	3378	539K	1646	153	n/a	50K	516Mb	5.6Gb
AlmaLinux	28K	2011	294K	1743	182	<106M	6492	572Mb	4.1Gb

1.4. Lies, Damn Lies, and Statistics

You should be careful reading too much into the statistics above. All are taken from a ‘‘default’’ installations on a qemu virtual machine using 1Gb of memory, but what the operating system does by default varies a great deal. I used a KDE desktop for openSUSE, a GNOME desktop for Debian and Alma, and no desktop for Alpine and Slackware. OpenBSD and NetBSD come with simple GUI’s, which I used, FreeBSD and DragonFly BSD don’t. The package counts for Slackware and the BSD’s are 3rd party source projects, while the other Linux and Solaris distros have precompiled binaries, these numbers will be higher since a source project can often be compiled into multiple binaries. Minix and DragonFly delegate more of the traditional kernel tasks to userland, and Plan 9 is a highly parallelized system, which will result in a higher number of background processes. Plan9Port isn’t an operating system at all, but a collection of userland programs. Some of the information above is only approximate or fragmental. For example the source code count for Solaris systems only cover the base system, 3rd party software is not included. The source for AlmaLinux also only cover the BaseOS repository of Red Hat. If all 3rd party code were included in OpenIndiana and AlmaLinux, the statistics would be close to that of Slackware. There are other factors to consider as well. See appendix A for details, but for now it’s wise to heed the axiom: *careful when reading statistics.*

1.5. Defining UNIX

Throughout this article we will frequently refer to ‘‘UNIX’’, what pray tell is the exact meaning of such a word? Ah, the flamewars kindled by our innocent youth... *And the tongue is a fire, a world of iniquity: so is the tongue among our members, that it defileth the whole body, and setteth on fire the course of nature; and it is set on fire of hell,* to quote the old book. Way back in 1969 *UNIX* was a very specific operating system, but 50 years of history has muddled the concept considerably. Today ‘‘UNIX’’ can mean different things depending on your point of view:

Judicial

Technically UNIX™ is a trademark of the X/OPEN Group. In accordance with copyright law, only they have the legal right to brand beasts with this epitaph. Companies wanting such a mark on their prize bull must undergo a ceremony called *the X/OPEN interface-specification tests,* and pay the X/OPEN Group a handsome royalty fee. MacOS, AIX, HP-UX, Solaris and other commercial systems are trademarked as UNIX, while Linux, FreeBSD, Minix and other opensource systems are not. In practice though this trademark is just an expensive sticker for vendors, it has no relevance to anyone, except lawyers.

Historic

Unlike Linux, the BSD branch of operating systems share a rich history with UNIX, they evolved directly from this primordial ancestor way back in the 70’s, and there was much collaboration between the two projects. Commercial UNIX borrow heavily from BSD, and textbooks at the time referred to BSD as ‘‘Berkeley UNIX’’. But ever since the AT&T lawsuit in the early 90’s, the BSD folk have been very careful to respect the above definition of UNIX, and officially call their operating system *UNIX* -like*, but they are non the less proud of their historical and cultural heritage (or baggage as a Linux user might say).

Pragmatic

History and culture are poor definitions. For example, while it is true that Linux is not directly descendant from UNIX, it is nevertheless a UNIX *clone*, meaning that it was reverse engineered to look, behave and *smell* exactly like it. And whereas half of MacOS X source code is a direct copy paste of FreeBSD, the culture between the two communities are certainly different. MacOS X and Solaris are both trademarked UNIX™, while FreeBSD and Linux aren’t, even though MacOS is eerily similar to FreeBSD and Solaris similar to Linux. A pragmatist would say *If it looks like a duck, and quacks like a duck, it’s a duck!* Hence it’s all UNIX. We largely use a pragmatic approach in this article, but the exact meaning of ‘‘UNIX’’ will depend on the context.

---

*) read: UNIX **wink wink**

Plan 9, Inferno, Haiku, SerenityOS and Minoca are also discussed in this article, calling these systems UNIX would be more like calling a platypus a duck. Only an ignorant fool would do so, but then again, there *are* similarities...

1.6. Defining a Good Operating System

Another superb source for flamewars is the question *what is the best operating system?* Besides being obviously irrelevant (best at *what?*), the question quickly taps into a whole world of tear dripping Pride and Prejudice. Try going in to a British pub and ask *what's the best football team chaps?* and you will get a sense of what I mean. At the end of the day, there is no such thing as ''the best'' operating system. Different systems are needed in different circumstances, and for better or worse, the one you end up using is almost always governed by your environment and subjective preferences. For some the availability of high quality source code is paramount, while trivial things such as gaming is unimportant. For others, strange as it may seem, it is the other way around.

Increasingly computers are becoming mere consumer products. People just want to buy a shiny new gadget and have instant plug-and-play gratification. While convenient, materialism does have its drawbacks. The new thingy magingy can never have too much specs, it can never be too shiny or too new, and yet the enjoyment it provides is short lived and shallow. They may call it innovation, but the insatiable drive for more stuff does little more then increase volume. This drive is also apparent in the opensource world. Many distro hoppers have an unquenchable thirst after new ''features'', even though they never actually use them. Despite its name materialism is not bound to physical objects.

UNIX however was created by developers for developers, it is essentially the reverse of a consumer-centric system. Instead of giving the users a five star hotel experience, it hands you a toolbox and gets out of the way as you go about your business. The tools must be of good quality, but simple and honed to the task at hand, void of frills and nonsense. This philosophy is great for work, but terrible for entertainment. Which is why UNIX has never been a good *product*, even though it's arguably the only relevant operating system for serious work.

As you may have guessed already, the author of this article tend to adopt the classic UNIX perspective of what a ''good'' operating system is, alienating many of our readers no doubt. Hopefully, a few objective findings will still sift through, and you may even find some humor in the snarky commentary. In any case, if you do want to understand what the UNIX philosophy is all about, my recommendation is to read *The UNIX Programming Environment*, and experiment in a UNIX shell as you do so. This classic book from 1984 is still 99% relevant for modern systems, and provides a powerful practical evidence for the wisdom of the UNIX philosophy.

2. ANCIENT UNIX

The development of UNIX was quite accidental. Bell Labs, in conjunction with other big organizations, had been working on the Multics operating system for years. It was a hugely bloated system and Bell Labs eventually pulled out in disgust and vowed that it would never ever ever have anything to do with operating systems ever again. Unbeknownst to management however, Ken Thompson, one of the many free minded hackers at Bell Labs, wrote a tiny toy operating system on a cast of PDP-7 in 1969, a ‘‘castrated’’ Multics, or EUNUCHS was thus created. With the support of a small team of other enthusiasts, they tricked management into buying a bigger computer, a PDP-11, which he and his buddies could continue to experiment on. The first release of UNIX was in 1971. By the 6th edition, released in 1976, the system finally broke containment and spread outside the lab. The rest as they say, is history.

Early editions of UNIX ran exclusively on the PDP-11, this machine affected development in many subtle ways. Each program had limited runtime memory for instance, forcing the development of small utilities. It used a physical lineprinter as its terminal, encouraging low verbosity and limiting interactivity, and its painfully awkward keyboard no doubt contributed to succinct misspellings. Early development of UNIX was surprisingly dynamic, and the system changed radically between releases. The well known UNIX philosophy developed gradually and through practical experience. For example, pipes were not invented until the 3rd release, but not in the form we know today, that came around in the 4th release. Standard error came about later too, when they discovered that sending errors down a pipeline was generally a bad idea. Even manpages were not formatted with troff, and the kernel not written in C, until the 4th release of UNIX. Early systems did not have a programmable shell, but rather a command shell akin to CMD in Windows. If you wanted to do some quick and easy coding in those days you did it in C, as opposed to the nitty gritty of ‘‘real’’ programming done in assembly. Meanwhile the University of California at Berkeley, got hold of a UNIX copy and released its 1st version of BSD (Berkeley Software Distribution) in 1978. Much collaboration between these two projects and other universities ensued. By the following year, the 7th edition of UNIX was released, introducing innovations such as the programmable Bourne shell, `sed` and `awk`. UNIX was becoming seriously powerful.

Around this time lineprinters started to be replaced with computer screens, `vi` and `more` (not to mention `rogue`) from the BSD camp made use of this new capability. The first BSD releases were really just a set of addons to UNIX from Bell Labs, in fact about 20% of 1BSD and 2BSD’s source code was the text editor `vi`. But less then a year after the 7th edition of UNIX, 3BSD was released as a fully functional operating system for the new VAX machine. This release convinced DARPA to fund the project, and in the following years BSD introduced some highly influential technologies, such as a crash safe filesystem, virtual memory and TCP/IP (aka. ‘‘The Internet’’). Not just that, but the entire BSD ecosystem had a very strong influence on what was later to become the opensource community, version control and open governance came from the BSD camp for instance, in addition to a host of software that we now take for granted. Modern Ubuntu users can probably fire up an old BSD release in an emulator and feel quite at home, whereas the old UNIX systems from Bell Labs will feel decidedly more alien! As a rule though the BSD developers largely ignored the UNIX authors drivel about simplicity and what not.* To quote

*) To be fair, the UNIX authors largely ignored the users drivel about speed and robustness. Besides features, the BSD camp considered such things important.

Kernighan and Pike, Anno 1984: *As the UNIX system has spread, the fraction of its users who are skilled in its application has decreased. Time and again, we have seen experienced users, ourselves included, find only clumsy solutions to a problem, or write programs to do jobs that existing tools handle easily. Of course, the elegant solutions are not easy to see without some experience and understanding.*

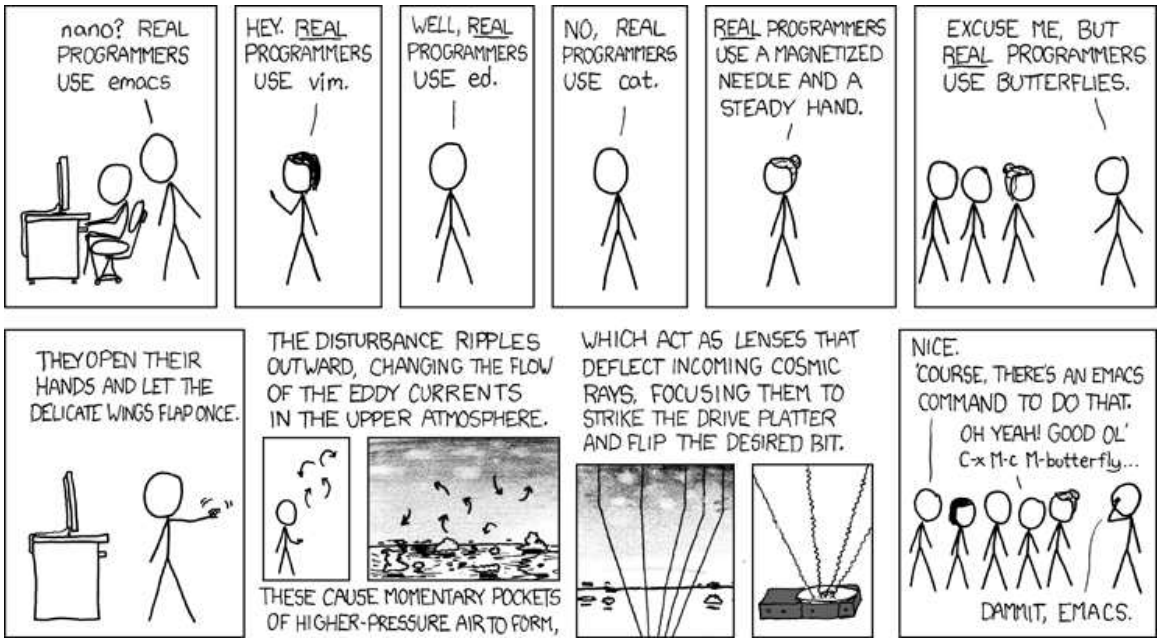
The growing interest in UNIX finally reached AT&T headquarters, the commercial empire who owned Bell Labs, by the early 80's, and the golden age was over. AT&T's first action was to make UNIX proprietary, and thus stop the proliferation of its source code, which more or less killed computer science. They consolidated the many diverse UNIX distributions that had sprung into existence into one proprietary system known as System III, and later System V. By the time that System V Release 4 (or SVR4) came out in the late 80's a great many commercial vendors were shipping their own brand of UNIX using AT&T's system as their base. Although a few of these commercial systems have survived down to this day, such as AIX and HP-UX, most of them died during the fierce competition from Microsoft in the 90's.

The original authors of UNIX were graciously allowed to keep working on their pet project, from now on referred to as Research UNIX, but were otherwise ignored by management as per usual (naturally, these researchers were equally disinterested in management). Eventually these researchers were so fed up with the archaic limitations of the now 20 year old operating system, that they rewrote everything from scratch. Their new UNIX successor, released in 1992, was called Plan 9 from Bell Labs. They were still under the evil management of AT&T however, so they were not allowed to release their work under a permissive license. This doomed Plan 9 into obscurity, where it has remained down to this day, even though its ideas has influenced all modern operating systems and several opensource forks of it now exist.

The second thing AT&T did was to sue their BSD competitor. The court case lasting from 1992 to 1994 seriously damaged BSD's reputation, and may have indirectly contributed to the massive growth of Linux, released during this critical juncture. Eventually the lawsuit was rendered mute when it was shown that AT&T's commercial systems had freely used BSD code without giving the Berkeley developers any credit, and thus had violated licenses themselves. In the settlement that followed Berkeley agreed to remove a handful of files and released their final BSD system in 1994. This system, 4.4BSD-Lite, was not an actual working system, since it lacked a few vital parts, but it was used as a base for the community forks FreeBSD and NetBSD, which were already in existence at the time. Later, other BSD forks, such as OpenBSD and DragonFly BSD, sprang into existence. All of these forks exist and are actively developed down to this day, and the permissive BSD license has allowed these systems to be used in a great many commercial offerings, such as MacOS and Netflix. In fact nearly all major vendors today, from Microsoft's TCP/IP stack, to Androids userland, to the routers on your wall, have pieces of BSD code in them.

When AT&T stopped giving away the UNIX source code to universities after Version 7, Andrew Tanenbaum, a professor in the Netherlands, rewrote the operating system for use in his computer science courses. This mini-UNIX, Minix, was intended as an educational tool only, not as a general purpose operating system. Linus Torvalds however, a young university student from Finland, read Tanenbaum's book on Minix and was inspired to write his own version called Linux. Released in the early 90's with a permissive license, Linux quickly became the main vehicle for opensource development. Today 100% of the top 500 supercomputers in the world, and 2/3 of the internet, are running Linux. It is by far the most dominant opensource project today, and it is largely responsible for breaking the proprietary hegemony of Microsoft.

2.1. Text Editors



	V1	**V5**		**V6**	**V7**		**41BSD**	**V8**	**43BSD**		**V10**
name	src	src	man	src	src	man	src	src	src	man	src
ed	~1215	1627	4	1182	1537	6	1605	1639	1593	7	1626
ex/vi							14,204	14,473	15,013	2	16,316
sam								2694			8688
emacs									69,815	n/a	

I'll begin by quoting the first couple of paragraphs in Michael W Lucas book *Ed Mastery* (2018)*:

> Let me be very clear here: ed(1) *is* the standard Unix text editor. Dennis Ritchie, co-creator of Unix, declared it so.
>
> Who are you to argue with someone who can write a complete operating system without using a glass teletype?[1]
>
> Many young sysadmins naively hoist their pennants to defend overblown, overwrought, overdesigned text editors like ex, vi, or even the impossibly bloated nvi. A few are so lost as to devote themselves to turgid editors meant for mere users, such as vim and Emacs. This way lies not only appalling sysadmin skills, but an absence of moral fiber. As a sysadmin, you must have enough brain power to remember what you typed, to hold your own context in your head, and to truly commune with the machine on a deep and personal level.
>
> [1]) You know, a glass teletype. That toy that kids keep calling a ''monitor'', even though we all know monitors are reference speakers used in audio production.

* That is not a typo, his book is from 2018.

On a more serious note, the question of what text editor to use was a hot debate even as early as UNIX V7. `vi` and Emacs were popular, so called *screen* editors, although by default the system only came with `ed`. `ed` was an old editor even then (in fact it even predates `echo` and C!), and was designed for a computer using line printers as terminals. Consequently `ed` sessions are non-interactive, that is, you type in whatever commands you need, while getting virtually no feedback. `vi` users can think of it as running `cat << EOF > file`, but with the added ability to type `ex` commands. To demonstrate a simple editing session:

```
ed greeting.txt
a                   append text
Hello World!
.                   stop appending text
w                   write file to disk
q                   quit
```

Of course a more realistic first `ed` session might look more like this:

```
ed

?
help
?
?
?
quit
?
exit
?
bye
?
hello?
?
eat flaming death
?
^C
?
^C
?
^D
?
```

`ed` wasn't a popular editor, despite the consistent and elegant user interface. Newbs continued to whine despite sysadmins loving advice, such as *read the manual*. Tough love it may be, but it's actually good advice. The 10 minutes it takes you to read the ed manual is time well spent, since many of the editors conventions are used throughout the UNIX system. And even if `ed` may not be well suited for interactive sessions where you need to jump back and forth in a file, it excels at batch jobs. You cannot control `vi` or Emacs from within a pipe, but you absolutely can use `ed` in this fashion. And unlike `sed` or `awk` it doesn't apply commands to every line in the file, instead you must specify which part of the file you want to change, this makes it much easier to work with isolated segments and blocks of text. For example to delete just the first block of text in a file:

```
echo '1,/^$/d
wq' | ed - file
```

This short example illustrates one of UNIX's most powerful ideas, ie. if programs take nothing but plain text as input (and output), it can become input *agnostic*. `ed` does not know, nor care, if its input is given by a human punching away at the keyboard, a file on disk or some other program writing to it. Of course not every program can be written in this way, but in UNIX this design *should* be the rule, not the

exception (in modern UNIX it is not). When this is the case, the whole system becomes transparent and scriptable. It takes some experience to realize this, but when this idea is followed through, even a simple operating system becomes incredibly powerful!

We have mentioned `vi` several times in our brief discussion already. This classic UNIX editor originated from the BSD camp, and was essentially `ed` with a *visual* interface, hence its name. In fact the termcap backend needed to support this visual text editor, that Bill Joy wrote around 1977, became the basis for all terminal pseudo-graphics down to this day! We can illustrate the similarities between `vi` and `ed` by writing the above example in `vi`:

```
vi greeting.txt
a                   append text
Hello World!
<ESC>               stop appending text
:w                  write file to disk
:q                  quit
```

The difference here, beyond the visual aspect of `vi`, is that the escape key is used to terminate text input, rather then a dot, and that some commands must be prepended by a colon. A rule of thumb here is that if the command in question is used to move about or shuffle some text on the screen, you don't need to prepend a colon, but if the command effects the whole file, such as saving, quitting or search and replace, you do. As you can see, `vi` requires the user to know many one letter commands for basic text operations just like `ed` does, but the situation is compounded by that fact that you need to learn a host of interface commands as well. Whereas `ed` has two modes, command and input, `vi` has four. It has been suggested (jokingly?) that a good random generator would be to activate a keylogger whenever a newbie tries to quit `vi`, since he will likely try out many different combinations before he stumbles on the correct sequence, `:wq` Yet despite the steep learning curve, `vi` quickly became a very popular editor and superseded `ed` as the de facto standard. Today you will find `vi`, or variations thereof, on virtually all UNIX-like operating systems.

The other popular editor of choice was Emacs, which originated from the Lisp machines at MIT. Lisp is a fascinating and unusually dynamic language, programs written in it are essentially self-hosting virtual machines. Whatever good things one can say about the language, and there are a great many to choose from, it does not generally follow the UNIX philosophy of tiny isolated programs that do only one thing. For Emacs this is especially true, it is one massive program that tries to do everything. Not only can you read your email, browse the web and listen to music on modern versions of GNU Emacs, but it even comes with tetris and a psychoanalyzer! Hence the old saying: *Emacs is an operating system, that only lacks a good text editor**

The authors of UNIX were aware of the interactive limitations of `ed`, but `vi` and Emacs did not peek their interest. The reason as Rob Pike explained it, was that it made no sense to add interactivity to a text only interface. Later when Research UNIX developed a graphical interface, Pike wrote the text editor `sam`, which is essentially `ed` plus a mouse driven GUI. The difference between `sam` and `vi`, is that the interface to `sam` is natural and easy to learn, there are no weird keyboard shortcuts, you just point and click. Later on Pike wrote another editor called `acme`. As the name suggests `acme` was designed to do-it-all. But unlike Emacs, it doesn't utilize a vast number of internal tools, instead it ties together the external tools UNIX already provides. Lastly, although these programs are graphical and interactively used by the mouse, they can be controlled by writing plain text strings to files, and are thus fully scriptable. We will discuss these text editors in greater detail in the Plan 9 section. But for now we can make an astute observation: While most UNIX users just assume that graphics and interactivity necessitates a deviation from the UNIX design philosophy, this is evidently not the case.

---

*) (this is false (you can run vi in Emacs (and ed in vi (and cat in ed))))

2.2. Internet

	V1	V5		V6	V7		41BSD	V8	43BSD		V10
name	src	src	man	src	src	man	src	src	src	man	src
mail	n/a	236	0.5	238	511	1	744	1039	663	9	12,031
write	n/a	196	0.5	197	168	0.5	188	296	227	0.5	315
news								293			235
telnet									1866	5	651
rsh									451	3.5	129
rcp								344	636	1	529
ftp									4532	9	4419
ifconfig									413	2.5	
sendmail									20,634	4.5	

The internet as we know it did not come into existence until the late 90's, and naturally the early versions of UNIX did not come with a web browser. TCP/IP, the protocol that makes the internet possible was introduced in 4.2 BSD in 1983, but communications between users and machines existed way before that, in fact UNIX was a multiuser environment from day one.

In the first chapter of *The UNIX Programming Environment* (1984), this demonstration is given:

> *Establish a connection: dial a phone or turn on a switch as necessary.*
> *Your system should say*

`login: you`	*Type your name, then press RETURN*
`password:`	*Your password won't be echoed as you type it*
`You have mail.`	*There's mail to be read after you log in*
`$`	*The system is now ready for your commands*
`$`	*Press RETURN a couple of times*
`$ date`	*What's the date and time?*

```
Sun Sep 25 23:02:57 EDT 1983
$ who                                   Who's using the machine?
jlb     tty0    Sep 25 13:59
you     tty2    Sep 25 23:01
mary    tty4    Sep 25 19:03
doug    tty5    Sep 25 19:22
egb     tty7    Sep 25 17:17
bob     tty8    Sep 25 20:48
$ mail                                  Read your mail
From doug Sun Sep 25 20:53 EDT 1983
give me a call sometime monday

?                                       RETURN moves on to the next message
From mary Sun Sep 25 19:07 EDT 1983     Next message
Lunch at noon tomorrow?

? d                                     Delete this message
$                                       No more mail
$ mail mary                             Send mail to mary
lunch at 12 is fine
ctl-d                                   End of mail
$                                       Hang up phone or turn off terminal
                                        and that's the end
```

There are many similarities between this workplace in the early 80's and modern ones today. Late hours at the office doing frivolous work seems to be a universal constant. But there are notable differences too. For one, this example demonstrates how much easier it was to collaborate with colleagues back then. At the time a university or company usually only had a single computer, a big mainframe hidden away in the basement. Multiple terminals were connected to this machine, perhaps one for every office. Being on a single computer allowed users to share files easily and delegate access to common projects using only basic commands, such as `cp` and `chmod`. `mail` only required a username as argument, `write` allowed you to chat one-on-one, `news` provided a common bulletin board, and there were other tools as well, `wall`, `who` and `finger` are some examples. Collaborating on this level with your colleagues today is non-trivial.

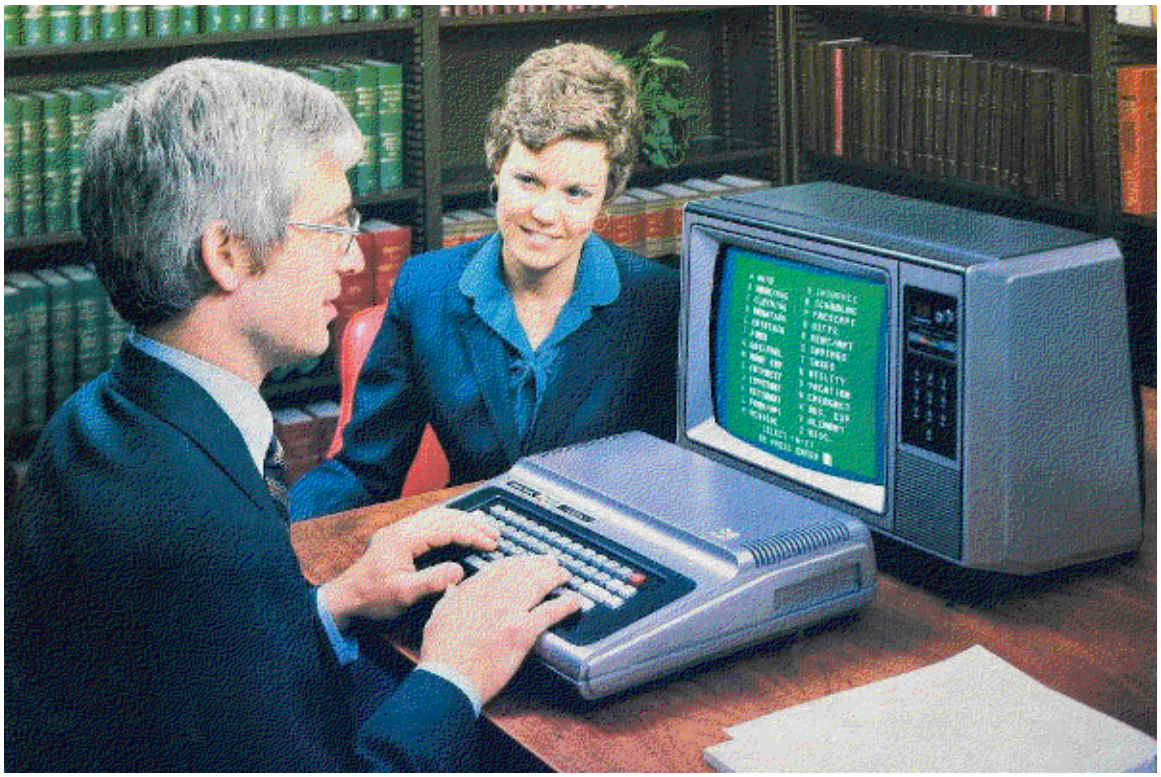
Also since the terminals were diskless, you only had to flip the power switch on, and log in with your user name and password to enter the system, to leave just flip the power switch off. And just like the university may have had only one janitor to take care of the physical buildings, it would only needed one sysadmin to maintain the computer.* The other employers could freely use the operating system without needing to know the finer details of how to maintain it. In fact, the secretarial staff at Bell Labs would commonly log in to a session that only ran `ed`. They did not need to know *anything* more about the system (although a shell escape was later built into the editor in case they did). Today everyone is running their own machine, and they all need to be computer experts to use it well, they all need to install their own software, monitor their own disk usage, update their own system, and fix their own problems as they arise.

The tools used for collaborating with colleagues today are huge and complex, since everyone has their own favorite chat program, and what not, which naturally support audio, video, emojis and any other conceivable barrier to thoughtful communication, which needs to navigate through all kinds of remote protocols, negotiate between layers of security, and desperately try to be compatible and ‘‘integrated’’ with every manner of platform and nonsense on the planet. It doesn't help either that everybody changes their hardware like underwear, making it impossible to learn, let alone standardize, anything. To quote the profound philosophical words of Rob Pike *it's just the stupidest idea ever.*

---

*) The difference being that when the sysadmin died, he would take the keys with him to the grave. Remember to backup your sysadmin folks!

2.3. Office



It may come as a surprise, but office utilities were a major part of the old UNIX system. In fact the official reason for funding the project in the first place, was to make a system that could typeset documents and thus save Bell Labs a great deal of money. The tool for this job was `troff`, based on `roff`, based on `runoff`, which ran documents off a printer. `troff` in turn inspired `tex`, which in turn inspired `latex`, which in turn has nothing to do with leather.

`troff` was used to write the UNIX manpages, and in this capacity it has survived down to our present day in various forms. But it's worth mentioning that within Bell Labs `troff` was used much more extensively, several high quality books and research papers, as well as regular correspondence in the lab was exclusively written in `troff`. It is a shame that the broader community didn't appreciate the capabilities of this tool, since it's such an elegant and capable markup language. Nevertheless `troff` isn't quite dead yet, the textbook *The Design and Implementation of The FreeBSD Operating System* (2015), was written in it, for instance.

	V1	**V5**		**V6**	**V7**		**41BSD**	**V8**	**43BSD**		**V10**
name	src	src	man	src	src	man	src	src	src	man	src
troff	n/a	2340	1.5	7687	12,922	1.5	13,408	15,241	11,617	1.5	31,750
(prep)				306	8553	4.5	4177	20,567	10,174	5	33,303
spell					634	2.5	683	1255	705	1	1622
diction							7541	2598	7706	1	7915
pr				403	381	1	391	465	410	1	453
lpr		182	0.5	218			1015		10,800	2	1574
fmt					470	n/a	498	747	398	0.5	182

For its time `troff` was a very large program, by V7 the `troff` suit of applications were about 15 times larger then the systems text editor. (by comparison modern `texlive` is 200 times larger still and takes at least half a Gigabyte of harddisk space) Over time a collection of preprocessors were created for `troff`, such as `tbl`, `eqn` and `pic` for tables, math equations and pictographics (the preprocessors are

collectively called ‘‘prep’’ in the table above). In addition to multiple macro packages, this made it potentially hard to compile a `troff` document into a readable format. For example, to read a documents with both graphs and tables in a terminal, you might have to type the ungainly command `grap doc.ms | pic | tbl | nroff -ms`. No doubt this is one of the reasons why it didn’t catch on. On the other hand, it is easy enough to automate this task with a script, and later Research UNIX and Plan 9 included a script called `doctype` that did just that. But the UNIX community outside Bell Labs ignored these advancements, and preferred the dubious elegance of even more convoluted office alternatives, such as Tex and DocBook.

The high focus on documentation at Bell Labs may have been somewhat of a happy accident. As mentioned document processing was a real need and persuaded management to fund the project, and since this was academia, not business, the researchers had ample time and resources. It can be safely said that few operating system have shipped with better documentation then UNIX V7. The manuals came in two parts, volume 1 was the technical reference (aka. ‘‘manpages’’), volume 2 was a collection of abstract papers that described various aspects of the system, such as practical tutorials, comments on security and so on. Volume 1 was indispensable for experts who already know the system, volume 2 was indispensable for people who didn’t. Volume 2 was twice the size of volume 1, and combined they had about the same amount of text as the system source code (just try to imagine what that would entail on a modern UNIX system!). In addition to that the system came with an interactive tutoring program called `learn`, which step by step taught you how to use the command line utilities, how to write documentation and how to program, all the while giving helpful advice for common mistakes. Lastly the overall simplicity of the system cannot be overstated. You could read *all* the documentation within a week, and even if you continued with the source code, you would be done well before the end of the month. No wonder that high quality documentation were produced at this time, when the system itself provided such clarity!

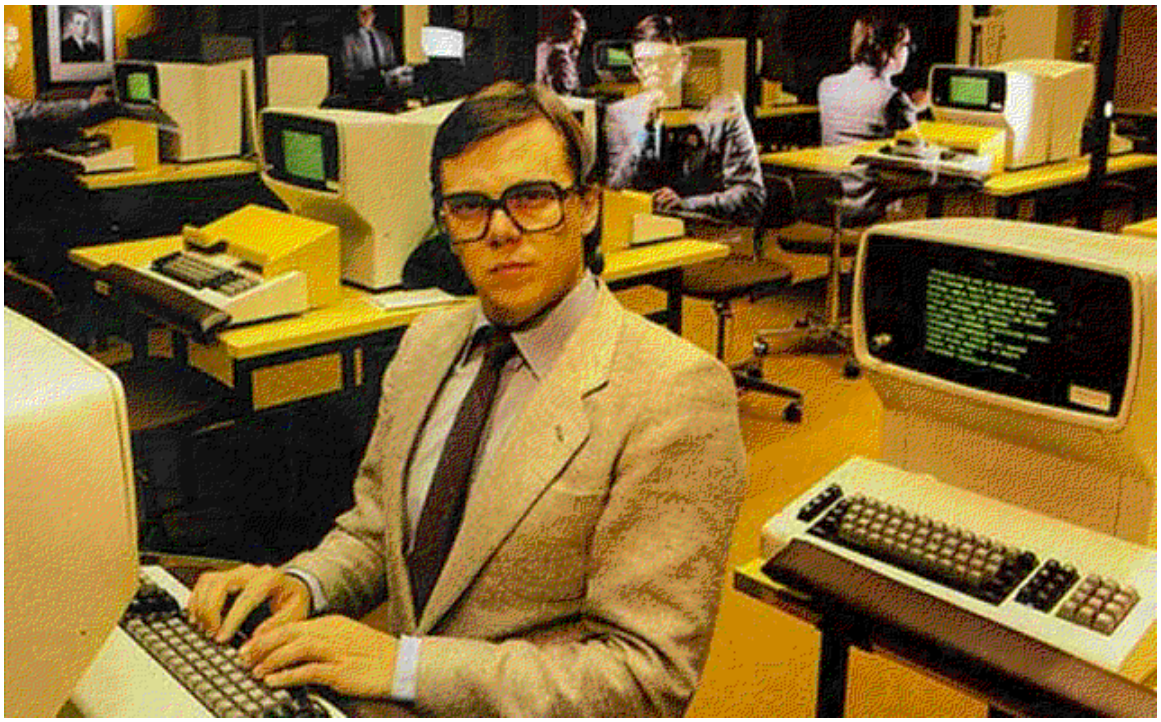
Some examples of other early office tools in UNIX include `pr` and `lpr` to paginate and print documents. `style` and `diction` (called `wwb` in V8), were used to analyze the text and look for bad prose and grammatical errors. Variants of these tools have survived down to this day, but are not much used, since, like, noone wood writes good prose anyways, nowadays. And of course there was `spell` for spell checking, a problem that is surprisingly hard to solve. The original UNIX spell checker was actually a shell script, which in modern syntax would look like this:

```
deroff $* | tr A-Z a-z | tr -c a-z '\n' | sort | uniq |\
comm -13 /usr/share/dict/words
```

The *words* file here is a database of correctly spelled words, one per line. The `look` command searches this file, and is a quick way to check for the correct spelling of a word. The main problem with the quaint script above is that it prints spelling errors for every word that doesn’t match the dictionary exactly. So if ‘‘computer’’ is in the dictionary, ‘‘computers’’ and ‘‘computing’’ will not be recognized as correctly spelled words, unless they also are added to the dictionary. Eventually a more sophisticated spell checker was written in C, handling simple grammatical prefixes, but this version has problems as well. The implementation is not as simple, it requires a binary dictionary which cannot easily be altered. And even though the implementation may handle English, it is not well suited for other languages. As time went on, even more elaborate solutions were made, deepening the problem further. For example, right-clicking a misspelled word in Microsoft Office and selecting the correct spelling in a drop-down menu, doesn’t actually help you learn correct spelling, it just trains your brain to click a mouse button. And when Facebook messenger auto-corrects ‘‘Noo’’ to ‘‘Moo’’, in response to your girlfriend asking if she is fat, it can create all sorts of problems. However simplistic the original UNIX spell checker may have been, at the very least it wasn’t detrimental to romantic relationships.*

*) Using ancient UNIX today on the other hand **is** detrimental to romantic relationships, but that is another story...

2.4. Shell



	V1	V5		V6	V7		41BSD	V8	43BSD		V10
name	src	src	man	src	src	man	src	src	src	man	src
osh	374	757	3	795	745	n/a					
sh					3177	7.5	3147	5192	3249	7.5	5861
csh							10,075	9395	9874	25.5	

The shell is arguably the main UNIX application, before GUI's it was the only interactive interface to the system. But more then that, the whole UNIX philosophy, and the rationale behind its design principles, are all centered around the shell.*

The idea behind UNIX is basically that programs shouldn't try to do everything, instead they should focus on doing a single task. In order to complete complex tasks in such an environment you need to use several programs in various combinations. The method for making these programs cooperate with each other is simple, just read and write nothing but text. If the program needs to have the text formatted in a certain way, the user can easily make any necessary adjustments with standard tools such as `sed`, `awk`, `grep`, `sort`, etc, or a plain text editor for that matter. In this way the user doesn't need to have special knowledge of ABI's and binary formats, he doesn't need to recompile programs or follow some strict inter-process mechanisms, he only needs to know what text goes into and out of the program. Not only is this approach simple, it turns out to be extremely powerful. Any requirements for making programs inter-operate are just conventions, often created by the users themselves. The system does not enforce any restrictions, and therefore there is no theoretical limit to what a UNIX program can do.

A practical example of this is the spell checking program provided in the previous section. Another classic example is this basic, text only, alternative to `tar`:

*) It is theoretically possible to follow the UNIX philosophy also for the GUI, but this has only been done in Plan 9.

```
echo '# To unbundle, sh this file'
for i
do
        echo "echo $i 1>&2"
        echo "cat >$i <<'End of $i"
        cat $i
        echo "End of $i"
done
```

Many inexperienced users will run the `echo` command for the first time and think ‘‘how is this useful?’’ Like hitting a nail with a hammer for the very first time, it may not seem all that useful, but really, that is the first step in building a house. A UNIX system is what you make of it. For example, waste time lamenting over the fact that the `rm` command actually *removes* files. If you want to move files to the trash bin, just do so: `mv file ~/trash`. Don’t fight the scary silence of `cp` and friends, relish in it. Peace of mind and an once of boredom go hand in hand with creativity.* The above example illustrate how easy it is to create new functionality with very basic UNIX tools. `tr` and `comm` were not created for the purpose of spell checking, and `echo` and `cat` certainly weren’t developed in order to replace `tar`. When the UNIX developers saw users doing things with their programs that they themselves had never envisioned, they considered it definite proof that the program had been successful (most other vendors have the exact opposite attitude). The role of the shell in UNIX is as vital as it is simple: provide the necessary glue between programs. Unfortunately since the shell was also the only interface back then, there was a strong desire to add interactive features to it, even though a text based interface is inherently unsuitable for such purposes.

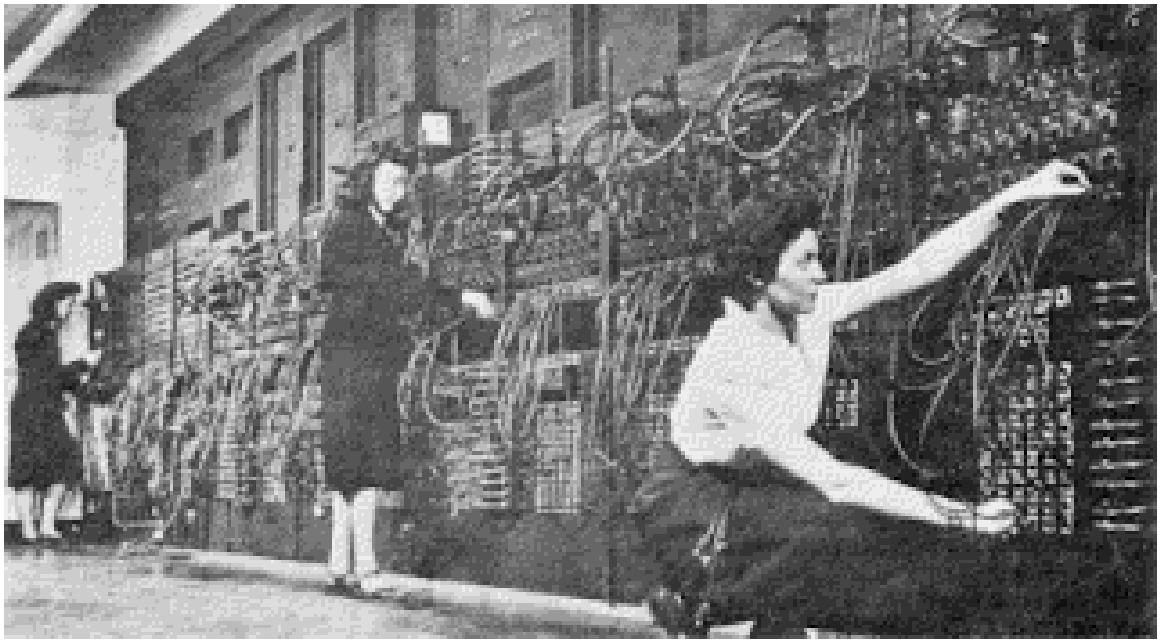
Early UNIX editions used the Thompson shell (referred to as ‘‘osh’’ in the table above). This was a basic command shell. It could do redirections and simple regex, rudimentary control flow could also be done with external commands like `if` and `goto`, but really, it had little functionality beyond executing programs. The BSD camp developed the C shell (or `csh`) early on, to address the lack of interactive features. It championed job control, history, substitutions and many other features that are commonplace today. Finally in Version 7, the UNIX developers from Bell Labs, replaced the old Thompson shell with their new Bourne shell. The new `sh` did adopt some features from `csh`, but their focus was not on interactivity, but rather to produce a solid programmable shell. The C shell was initially very popular, but it wasn’t superb for scripting purposes, and slowly descended into obscurity as newer variations of the Bourne shell incorporated its interactive features. Today virtually all UNIX users, except for the most ardent FreeBSD Luddite, use `sh`, or a descendant thereof.

The Bourne shell was a solid upgrade from the Thompson shell, but it did have flaws. Unlike `csh` which mimicked the syntax of the new and obscure C language, the Bourne shell mimicked the well known ALGOL language from the 60’s, to make it easier on the newbies. In retrospect that wasn’t so brilliant. Sensibly the shell has only one datatype, but it’s unfortunately a string, making arrays difficult. In addition there are three escape characters with horrifically complex expansion rules. Commercial UNIX later developed the Korn shell, which enhanced the Bourne shell in many ways, but backwards compatibility prevented it from addressing many of the underlying problems, and compounded the ugly syntax even more. The opensource `bash` shell from the Linux camp later compounded the situation still. When Plan 9 was developed, the original UNIX authors decided to rewrite the shell without concerning themselves with backward compatibility. The resulting `rc` shell has a C-like syntax and uses lists of strings as its datatype, making arrays seamless, and single quotes are the only escape character. Unlike modern UNIX shells such as `bash` and `ksh` it does not support math or interactivity. Other tools provide that, so adding them to the shell would serve no purpose other then thickening the manual. The shell was also ported to V10, and later to BSD and Linux. It should be pointed out that although UNIX provides a programmable shell, the shell wasn’t meant to be a fully fledged programming language. It is merely a glue between programs, with just enough features to automate your everyday chores. Whereas the `rc` manual has less then 9 pages of text, `bash` has nearly a 100. With the amount of effort required to learn `bash`, you may as well learn a real programming language.

---

*) It is possible to write a program that gives you a progress bar if you want it (see the `pv` command).

2.5. Applications



	V1	V5		V6	V7		41BSD	V8	43BSD		V10
name	src	src	man	src	src	man	src	src	src	man	src
echo	n/a	9	0.5	9	21	0.5	22	74	22	0.5	78
cat	97	58	0.5	59	56	0.5	132	55	194	0.5	58
ls	608	394	1	420	380	1.5	1382	651	616	2	595
find	n/a	417	1	404	681	1.5	673	651	1232	2	672
cp	56	71	0.5	51	83	0.5	106	79	211	0.5	281
wc	n/a	48	0.5	68	78	0.5	178	124	n/a	0.5	106
sed					1612	2.5	1619	1532	1625	2.5	1633
awk					2577	2	2781	3576	2834	2	4484
ar	1944	507	1	507	617	1	647	648	668	2.5	667
tar					842	1.5	862	915	1210	2	1000
sort	n/a	550	1	549	830	1.5	841	962	841	1.5	1140
tail					168	0.5	187	204	202	0.5	290
ps		177	1	264	361	1	995	727	1508	3.5	989
file				224	301	0.5	354	396	459	0.5	505
grep		463	0.5	298	1280	1.5	806	1304	1381	1.5	956
date	125	160	0.5	134	148	0.5	149	142	303	1	170
bc				519	558	2	564	575	566	2	581
(pager)							1395	385	1553	3	365

As you can see, many of the classic UNIX utilities were available early on, even for Version 1 (although they behaved differently since pipes had not yet been invented...). Version 7 was the last edition to mainly use a physical lineprinter as its terminal, and as such it did not come with a pager or a visual text editor by default. The pager `p` and the graphical text editor `jim` (a precursor to `sam`) were introduced in the 8th edition of Research UNIX, but the BSD alternatives `more` and `vi` were much more popular, so

much so that many people today mistakenly assume that these programs originally came from UNIX (other surprising tools that originates from BSD are `head`*, `telnet` and the lisp and pascal ports).

From the Bell Labs developers perspective these programs were too complex and too poorly designed. As with `vi` and `csh`, `more` implements many interactive features (and `less` even *more* so). It was understandably tempting to add interactivity to user applications when early UNIX didn't have an interactive desktop, but it was nevertheless a mistake to do so. It massively bloated these programs and distracted developers from their main purpose, it also created a plethora of inconsistent pseudo-GUI's to the consternation of its users. The correct solution is to create one interactive program, a GUI that integrates well with the terminal, let this program take care of interactivity, and let the other applications focus on their main task. This way the programs remain lean and simple while the user only has to learn one interface. The original UNIX developers understood this well, and followed this approach as graphics became an option, the community at large did not.

It is interesting to compare early UNIX applications to their modern counterparts, even simple programs such as `echo` or `cat`, have grown surprisingly fat on most modern UNIX editions. In fact the proliferation of new features in `cat` motivated Rob Pike and Brian Kernighan to write the classic essay *cat -v considered harmful* (it is well worth a read). Some classic UNIX tools are blown totally out of proportion, comparing V8 to FreeBSD for example, `tar` is 9 times bigger, the pager 50 times and `file` 160 times bigger! These three examples illustrates nicely what happens when you: `tar` - add too much functionality, `less` - add too much interactivity, and `file` - do not follow the principal ''worse is better''. These problems are expanded upon later in the BSD chapter.

Nevertheless some classic UNIX tools haven't actually changed that much, such as `sed`, `awk` and `grep` (''grep'' here refers to all three implementations, `grep`, `fgrep` and `egrep`). `awk` was first introduced in V7, and later extended in V8. Since then this second edition has been the de facto standard. Some commercial UNIX brands, such as Solaris, include both editions and refer to the second as `nawk` (new awk). Of course when I say that these tools haven't changed much, I am *not* talking about GNU (ei. GNU/Linux)! The GNU editions of even the simplest UNIX utilities are bloated far beyond mortal understanding. The GNU version of `echo` for instance, is 212 lines long, has 5 include statements and 3 goto's. In contrast V5 `echo` looks like this:

```
main(argc, argv)
int argc;
char *argv[];
{
        int i;

        argc--;
        for(i=1; i<=argc; i++)
                printf("%s%c", argv[i], i==argc? ´\n´: ´ ´);
}
```

The `/usr/src/cmd` directory in V7 holds about 160 utilities, which have a combined source code of nearly 130,000 lines, and section 1 of the manuals number 130 pages. As for V8 the directory holds about 290 programs (of which about 190 are user utilities), with a combined source code of 360,000 lines, and section 1 of the manuals number 230 pages. BSD 4.3 has about the same stats, as had commercial UNIX at the time. These numbers are fairly comparable to Plan 9, but are very frugal when compared to modern UNIX systems, which usually have two or three times as many utilities, and twenty times the documentation! (or in the case of Linux twenty times the source code and no documentation) Sysadmins at the time had printed editions of the manuals in their bookcase, and they actually read them. The definition of a ''guru'' is one who can read a manual, as the old joke goes, but it's worth mentioning that sysadmin in the olden days actually *could* read the entire manual. This is not possible today as the manual is literally a 1000 times thicker.

---

*) According to Rob Pike, because the BSD developers didn't know about `sed 10q`

2.6. Desktop

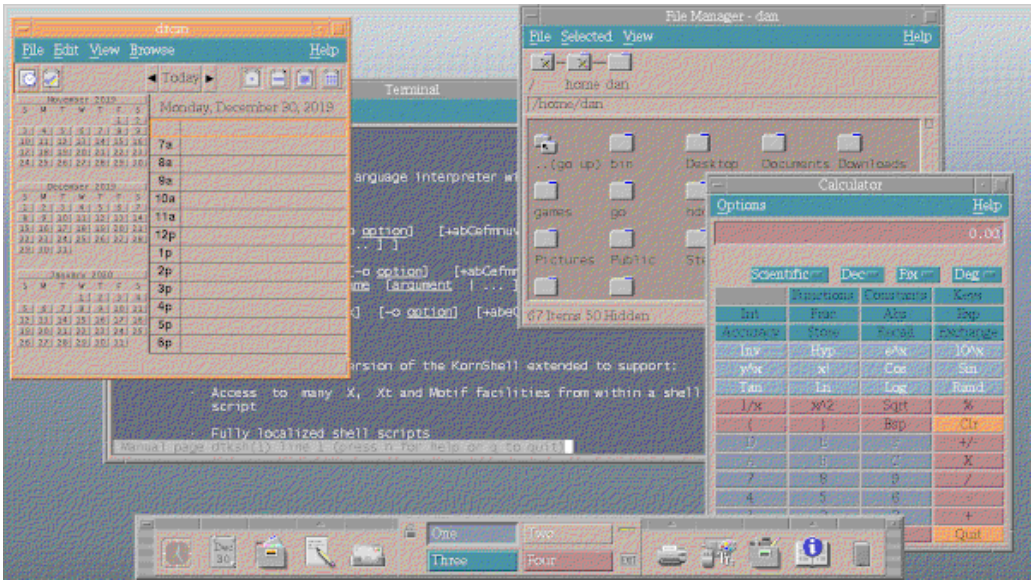
The original UNIX ‘‘desktop’’ was a lineprinter. These were replaced in the 80’s with screen teletypes, essentially what we call ‘‘terminals’’ in modern UNIX. In the early days a teletype could only run one terminal, which is why job control in `csh`, and window management in Emacs, were such a big hit at the time. The advancement from screen terminals to graphical systems was fairly quick. Already in the late 80’s commercial UNIX was tinkering with what later became X, and by the early 90’s the fully fledged desktop environment CDE was developed. The opensource alternatives took a few more years before they reached the same level of maturity.

Meanwhile the original design team at Bell Labs were, predictably, not impressed. X was modeled after desktops of non-UNIX systems, which firstly, did not understand the significance of text, and secondly, were excruciatingly convoluted. Already by Research UNIX V8, the team had developed a graphical terminal called Blit. The concept was similar to the old mainframe approach, a powerful text-only server was installed in the basement, and connected to it were a number of low-powered diskless terminals. The difference was that these terminals could run a simple windowing system. This window system later evolved into the Plan 9 desktop `rio`, which is about 300 times smaller then CDE.

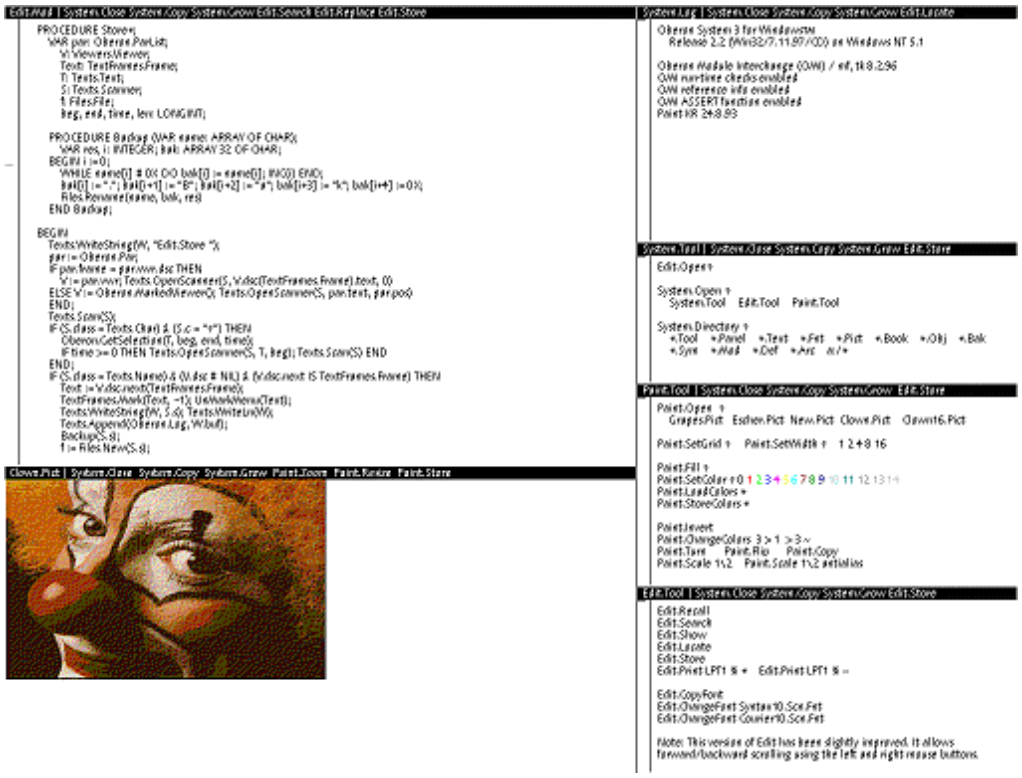
The difference in focus between these two developments of graphics were paramount. While X tried to develop a massive Windows like system full of new GUI programs that each tries to do everything, with zero collaboration with existing programs. Blit was designed for the purpose of multiplexing terminal windows. The original UNIX team exploited graphics in many interesting ways to augment the text terminal. For example while X went to great lengths to emulate the physical limitations of teletypes in `xterm`, Blits terminals behaved much like a regular GUI text editor. You can freely copy-paste and edit text using simple mouse actions, like you would in any graphical editor. This meant that many interactive features of the text terminal; substitutions, history, line editing etc, were unnecessary, and subsequently dropped.

One external desktop did peek their interests though, that of the Oberon operating system. At the surface the Oberon desktop looks like a regular tiling window manager, but its approach to GUI’s is radically different and unique. Text can be written anywhere inside the GUI and executed with mouse actions. This

design is simple and ingenious, any command line program is automatically available in the GUI, and tech support is simply a matter of emailing the correct instructions to the user and asking him to click on them. This design greatly inspired the `acme` text editor in Plan 9.

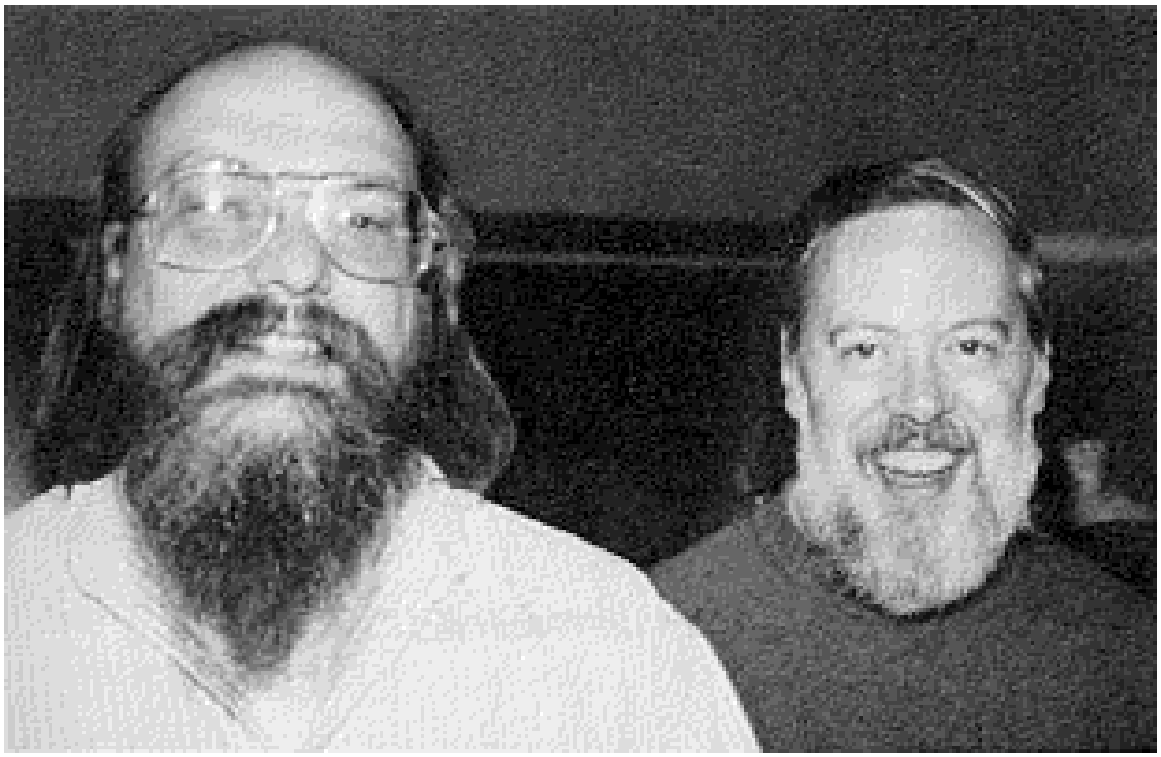


CDE Desktop



Oberon Desktop

2.7. Programming



Ritchie and Thompson, inventors of time (ei. 1 Jan 1970)

	V1	**V5**	**V6**	**V7**	**41BSD**	**V8**	**43BSD**	**V10**
name	src	src	src	src	src	src	src	src
as	˜928	3248	3249	3319	4365	4542	6163	4412
C	253	537	770	2586	9699	15,629	19,829	17,430
F77		433	431	10,377	10,456	13,430	22,190	24,538
Pascal					2700		39,874	
Lisp					52,312		59,186	

Naturally, C and UNIX are two birds of a feather, but the development of C, like UNIX itself, was gradual. The kernel wasn't written in C until V4. It's important to note here that the value of C back then was *not* efficiency, but ease of use. To quote Dennis Ritchie: *Early versions of the operating system were written in assembly language, but during the summer of 1973, it [ei. the kernel] was rewritten in C. The size of the new system is about one third greater than the old. Since the new system is not only much easier to understand and to modify but also includes many functional improvements... we considered this increase in size quite acceptable.* After the kernel had successfully been ported to C, the developers started to view the new language as a serious tool for system programming. The first edition of *The C Programming Language*, fondly referred to by fans as ''The Old Testament'', was published in 1978 for V6 of UNIX. Yet only 45% of the userland code in V6 was C, the remaining 55% being written in assembly. By V7 however 95% of the userland was written in C, the language, as the system itself, was starting to look like the UNIX we all know and love today. Some years later (1988), the second edition of *The C Programming Language*, aka. ''The New Testament'', describing the ANSI standard C, came out. It is the defining reference manual for the language down to this today.

V7 also shipped an industry standard Fortran (F77) compiler, and arguably decent shell scripting for the first time. Support for Pascal and Lisp were introduced in BSD at an early date, in fact the primary motivation for developing the ground breaking virtual memory for 3BSD, was so that it could run these memory hungry languages. By the late 80's C++, Perl, Python, Tcl and quite a few others had sprung up. Yet despite this wealth of options, there were, and is, only one true language in UNIX: C.

Like UNIX itself, C is largely misunderstood and under-appreciated. The purpose of C isn't to do everything, rather it tries to give you a simple and small language, that is fine-grained enough to manipulate the underlying hardware directly. To quote Kernighan & Pike: *C is a razor-sharp tool, with which one can create an elegant and efficient program or a bloody mess.* Higher functions must be supplied by the surrounding ecosystem, either as libraries or as system calls. Thus C can be a good language in a good system, but it will definitely be a poor language in a poor one. For example, `cp` on both modern UNIX and Plan 9 is written in C. In modern UNIX networking is implemented as a complicated sockets interface, and patching the C code in `cp` so that it can copy files over the network, is a non-trivial job. In Plan 9 networking is implemented within the filesystem, so `cp` can transfer files across the network without needing any additional changes. Whereas `cp` and `scp` in OpenBSD is 570 and 1364 lines of C code respectively, Plan 9's `cp` fills both these roles and is only 179 lines of code. (and OpenBSD has *good* C code compared to other modern UNIX's!)

The horrifying complexity of modern UNIX can be seen in many other areas as well, graphics and system administration are two obvious examples, `ps` in Linux is about 5000 lines of C, and X about 8,000,000. In contrast `ps` in Plan 9 is 158 lines and its window manager, `rio` is 5518 lines of code. The latter programs have readable source codes, the former do not, despite being written in the same language. The lesson here is simple, writing good code is not primarily a question of programming language. Support for modern features such as concurrent programming, unicode, networking, graphics and more, is poor and inelegant in modern UNIX, but very good in Plan 9, and their respective C source code reflects this reality. Popular programming languages, such as Python, are very good at shuffling all this mess under a carpet, giving programmers the illusion that things are simple. But the underlying operating system, not to mention Python itself, is still written in C. So a good Python programmer that doesn't know C, is quite ignorant about the world, however efficient he otherwise might be. Of course none of the complicated stuff mentioned above existed in the 70's, and programming C in ancient UNIX is both simple and elegant.

Shell scripting should also not be overlooked. Like C, it too is a very simple tool, that relies on the surrounding system for its power. You can do *much* more with the humble shell then you may think. If you supply your bookshelf with *The C Programming Language*, *The AWK Programming Language,* and *The UNIX Programming Environment* you are well set to develop useful applications in ancient, as well as modern UNIX.

2.8. Kernel

	V1	**V5**	**V6**	**V7**	**41BSD**	**V8**	**43BSD**	**V10**	
name	src	src	src	src	src	src	src	src	comment
boot					1635	4244	1774	1456	startup proc.
h				1209	3688	7231	3068	3845	headers
sys				4930	12,269	14,553	20,570	6156	kernel facilities
dev				6687	15,580	29,285	41,022	21,915	device drivers
net						4592	10,834	4195	network
(misc)				2788	2425	7512	11,013	18,962	miscellany
sum	3976	7261	9019	15,614	35,597	67,417	88,281	56,529	in total

A good text book on the early UNIX kernels is the legendary *A Commentary on The Sixth Edition UNIX Operating System* by Lions. Of newer era is *The Design and Implementation of The 4.3BSD Operating System*, and *The Design of The UNIX Operating System*, which focuses on the AT&T UNIX System V Release 2. Although over 30 years old now, these latter books actually describe operating system fundamentals that are still quite relevant today. And because they describe systems much simpler then our modern counterparts, they are comparatively easy to read and understand.

As an alternative, you can look into Xv6 from MIT. Xv6 is a rewrite of UNIX Version 6, using modern C and targeting the RISC-V architecture. Like Lions book, it has a line-by-line commentary of the code. The operating system is less practical then V6 was, but is much easier to learn, since you don't need a ton of historical context to understand it.

As you can see in the statistics above, the kernels for these ancient systems were tiny, one reason being that there were virtually no device drivers required. Prior to V7, UNIX only supported one machine, but even after that very few computers, with few devices were available. One of the greatest innovations of UNIX was to represent everything, including hardware devices, as files. Largely as a result of this innovation, modern UNIX systems can include drivers for virtually all hardware known to man in less then 100 Mb, a feat that would be totally unthinkable in Windows. As devices became more complex however, compromises to this design principal were made, such as ioctl, sockets and pseudo files, making modern UNIX systems that much more complex and obtuse. (although not analyzed in detail here, the commercial System V kernel from AT&T had about the same complexity as the contemporary BSD versions*) Many developers thought that such compromises were necessary to make modern hardware work, but that wasn't the case. Plan 9 demonstrated in practice that modern hardware can be represented as plain files, as long as the system is thoughtfully constructed, and not just haphazardly McGyvered together to run as fast as humanly possible. Presenting modern innovations such as graphics, audio and networks as plain files, dramatically increases the power of the shell and the C language, but we will talk more about this in the upcoming Plan 9 section.

To quote Kernighan and Pike, Anno 1984: *The UNIX system has since become one of the computer market's standard operating systems, and with market dominance has come responsibility and the need for ''features'' provided by competing systems. As a result, the kernel has grown in size by a factor of 10 in the past decade, although it has certainly not improved by the same amount. This growth has been accompanied by a surfeit of ill-conceived programs that don't build on the existing environment. Creeping featurism encrusts commands with options that obscure the original intention of the programs. Because source code is often not distributed with the system, models of good style are harder to come by.* These problems has persisted and deepened in modern UNIX, compared to V7, the Linux kernel has grown by a factor of more then a 1000, although it has certainly not improved by the same amount, and models of good style are increasingly hard to come by, even in systems where the source code *is* available.

Lets be clear here. Ancient UNIX was in no way a ''perfect'' system, it had bugs and warts all over the place. But it was a powerful and simple operating system. Refining and debugging, or for that matter rewriting, a system is all well and good, as long as you don't double the source code in the process. We may perhaps conclude with another quote from Kernighan and Pike: *One thing that UNIX does not need is more features. It is successful in part because it has a small number of good ideas that work well together. Merely adding features does not make it easier for users to do things - it just makes the manual thicker. The right solution in the right place is always more effective than haphazard hacking.* UNIX had 500 pages of manuals when they wrote this, today it has 50,000.

*) Hopefully that tidbit of information did not give away any important trade secrets.

3. PLAN 9 and INFERNO

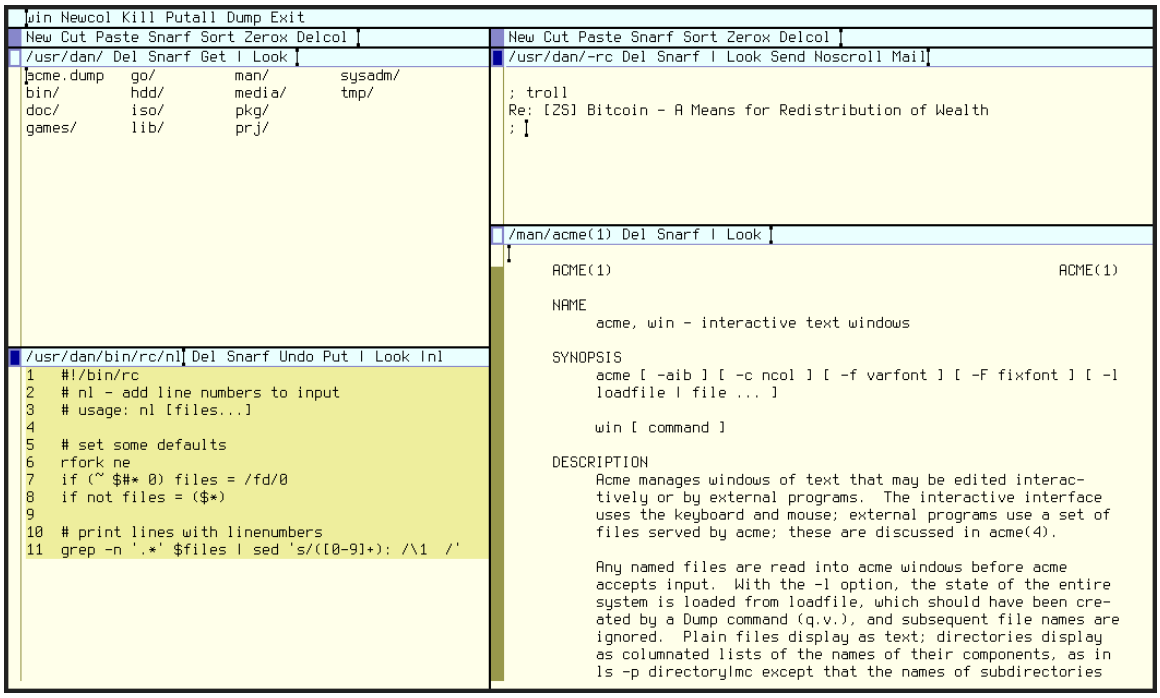
Plan 9 from Bell Labs, initially released in 1992 was created by the original authors of UNIX. It was a research operating system like its predecessor, written from the ground up with the aim of bringing the ideas of UNIX into the modern world of computers. Graphics, unicode, networks and multi-core processors did not exist when UNIX was first developed in the 70's. And the implementation of such things later did not follow the original design principals that these authors had set down. Thus they felt the need to rethink the operating system, to quote Rob Pike: *Not only is UNIX dead, but it's starting to smell bad!* Plan 9 demonstrated that a modern graphical and networking system was possible using the UNIX design philosophy (eg. everything is a file). Bell Labs discontinued the project in 2015.

Plan 9 isn't quite dead though, more like undead. Recently, The Plan 9 Foundation (`https://p9f.org`) was set up to continue development, but as of yet, little has happened. 9front and 9legacy are two of several community forks, that sprang to life around 2010, the former is radical in its pursuit of new features (such as hardware support), whereas the later tries to stay true to the original form like a stubborn caveman. We will also mention additional forks that are a bit unusual in our analysis: Harvey and JehanneOS are ports of the Plan 9 code to GCC, it is intended to be run on top of UNIX using the `qemu` virtual machine, but all its files are stored locally on the host. In theory this port could make it easier to add UNIX software to Plan 9, but it's early days and the projects aren't quite there yet. Meanwhile Plan9Port goes in the opposite direction, as the name suggests, it is not an operating system but a port of Plan 9 applications to UNIX. The project is both mature and practical, but limitations in UNIX does not allow for a full emulation of the Plan 9 experience.

Finally, Inferno was developed in tandem with Plan 9, by the same group of developers. Although many aspects of the two systems are similar, and they even share some code, Inferno had very different goals. It was developed as a commercial product and often compared to Java, since it too ran as a virtual machine and offered uniform programming resources on many different platforms. But Inferno goes beyond that and provides a fully functional operating system. It was written in an entirely new language called Limbo, a precursor to Go. The system had some very innovative ideas, even the name is quite providential, as it all went straight to hell due to damned finances. Bell Labs eventually discontinued the project

and abandoned it to opensource, where Inferno has remained in limbo ever since. Recently, the 9front developers have dug up the corpse and done some promising experiments with it. One of their forks, 9ferno, can now build on 64-bit architecture, and hopefully, we might see further improvements still. In this chapter we will use application statistics from 9front and 9ferno, but their numbers should be more or less identical to the classic Plan 9 and Inferno systems.

3.1. Text Editors



Acme the do-it-all application

9front			**9ferno**	
name	src	man	src	man
hold	5	0.5		
ed	1405	6.5	1391	n/a
sam	6236	8	2210	n/a
acme	12,664	8	14,378	12.5
wm/edit			617	~0
wm/brutus			3873	1.5

`ed` of course is the standard UNIX editor, in Plan 9 it has been modified to handle unicode and use the systems standard regex library, which is similar to that of `egrep` in UNIX. `sam` is essentially an extension of the `ed` command language plus a detachable GUI. It can perhaps be compared to `vi`, but the GUI is entirely mouse driven and considerably simpler.

`acme` on the other hand as the name suggests tries to do everything. It is used as a text editor and IDE, a terminal, a mail client and a file manager. The editor is heavily inspired by the Oberon operating system, where text can be typed anywhere into the GUI and executed with a mouse click. The idea may seem alien at first, but it is simple and intuitive once learned. The benefit of this approach is that it ties the editor seamlessly into the surrounding environment (the text mangling functionality of `sam` is incorporated in the built-in `Edit` command).

For example to select all the text in a file, type and middle click `Edit ,`. This is essentially the same as `:%` in `vi`. To get word count statistics of your file, just type and middle click `Edit ,| wc`. Or if you want to `rot13` some text, just select the text you want to edit, then type and middle click `| tr a-zA-Z n-za-mN-ZA-M`. I'm sure you can think of many other ways to mangle text using standard UNIX tools. But typing in long commands for routine operations quickly becomes tedious, so make a script for it, just as you would in a shell. The following `t+` script will indent the selected text for instance:

```
#!/bin/rc
# t+ - indent input
# usage: t+ <input >output
sed 's/^/       /'
```

With this in place you can indent text in `acme` just by selecting the text in question and then middle click `|t+`. In addition to having all textual programs instantly available in the editor, `acme`, like the window manager `rio`, and indeed Plan 9 itself, is actually controlled by writing text strings to a set of files. This allows you to easily script the editor itself. For example, if you wanted to clear a terminal window in `acme` of all text, you could manually write `Edit ,d` (that is `:%d` for all you `vi users out there`), anywhere in the window, highlight the text with your mouse and then middle click it. But we can do all of that automatically with a simple script:

```
#!/bin/rc
echo -n Edit ,d   > /dev/acme/body
echo -n /Edit ,d/ > /dev/acme/addr
cat /dev/acme/addr | awk '{ print "MX", $1, $2 }' > /dev/acme/event
```

MX here refers to a mouse middle click in the body of text, it is followed by the start and end location of the selected text, which `/dev/acme/addr` provides for us once we have written the regex we are looking for into it. Here is another example: Suppose we have written a text only slide show with files called `1`, `2`, `3`.., and so on. Using `acme` as a filemanager, we can right click our way to the slides and just click on `1` to open the first slide. We could in theory navigate to the next slides manually, by editing the ''1'' in our filename to ''2'', type the command `Get` and middle click it. We would have to do so twice, since `acme` will warn us about loading a file if the current content is different. This is cumbersome, so lets automate the process:

```
#!/bin/rc
page='{echo '{basename $%} + 1 | hoc}
if(test -f $page)
        for(cmd in 'name ''{pwd}^/$page'' clean get)
                echo $cmd > /mnt/acme/$winid/ctl
```

`$%` here is a special `acme` variable that refers to the current absolute filename, if for instance our file is `/usr/dan/lib/slides/2` then `basename $%` will give us `2`. `hoc` is a calculator, you can also use the old UNIX warhorse `bc` if you want. You'll note that the `acme` control files in this example is in a different location then the previous example. `/dev/acme` points to your *current* acme window, whatever that may be. `/mnt/acme/$winid` however points to a specific `acme` window using its unique ID number. The important point here is that each Plan 9 process has its own namespace, so `/dev/acme` for instance, can mean different things to different processes, but there are still ways to specify which process you mean if you need to. Anyway, the last for loop here sets a new name, tells `acme` not to mind the fact that the content is different, and finally asks it to load the new file. Assuming we call this script `Slide+`, all we have to do to step through our slide show is to middle click this command

We have really just scratched the surface with our examples here, I am sure you can think of more complex and useful scripts to write, an `awk` script to handle spreadsheets perhaps? Or maybe a script playing through, and otherwise manipulating, a playlist of audio files? It wouldn't be too hard to do. The crucial point to grasp here is that `acme` is seamlessly *integrated* with the environment around it. It does not have its own weird internal scripting language or obtuse ABI, you can script it using programs that the system already provides, writing and reading text to plain files. Just as the UNIX philosophy gives

tremendous power to shell utilities that follow its principals, so it gives tremendous power to GUI programs that adhere to it. It is relatively easy to extend `acme` with sophisticated applications. The fully functional `Mail` client for instance is just over 2000 lines of C. Yet all the power `acme` provides is combined with a remarkable simplicity, the manual is only a few pages long and the source code a hundred times less then that of GNU Emacs. Is it any wonder then that `acme` has been described as ‘‘Emacs done right.’’

Inferno also includes these editors, although it doesn’t seem like the `sam` port was fully completed. In addition it ships with the Notepad and Wordpad like `edit` and `brutus`, in all probability, to soften the blow for unsuspecting Windows users. `brutus` is a WYSIWYG SGML editor (ei. a ‘‘Wordpad’’ for DocBook), which actually can be kind of useful. Note also that creating Notepad in Inferno only requires 619 lines of code, in contrast ReactOS version of Notepad is over 8000 lines of C, and `leafpad` in Linux is over 18,000 lines. Speaking of simple text editors, 9front includes a very basic text editor called `hold`, here is the source code in its entirety:

```
#!/bin/rc
{
        echo holdon >[1=3]
        cat $1 > /dev/cons
        cat /dev/cons > $1
} >[3]/dev/consctl
```

The reason you can write such a ludicrously simple text editor in Plan 9 is because the terminal already behaves very much like a text editor, so the only thing you need to do is print the file and then save any text subsequently written afterwards on the terminal. (So to ‘‘edit’’ the file, copy paste it first, then make changes to the copy) True, it’s a bit clunky, but hey - it’s a text editor written in five lines of shell script!

3.2. Internet

9front			**9ferno**		
name	src	man	name	src	man
abaco	6375	0.5	charon	21,556	6
mothra	7274	2.5	collab	2676	3
upas	35,081	3	wm/telnet	715	0.5
ircrc	234	2	wm/readmail	788	-
ssh	1263	1	wm/sendmail	576	0.5
hget	105	2			
hpost	208	-			

The Plan 9 documentation mentions several places that internet browsers are a work in progress, and is essentially an unsolved problem. Things did not improve when the developers abandoned the operating system.* 9front ships with their rewritten `mothra` browser in addition to the classic `abaco` browser. Neither do anything except read plain HTML. `abaco` have been dropped in Harvey and Plan9Port, instead the `web` script in Plan9Port launches whatever default UNIX browser you are using. It would have been nice to have a web browser in `acme`, but realistically this isn't feasible. Although you can do a simple text dump of a web page, assuming of course the unlikely event that the web page in question is in readable HTML: `wurl2txt http://www.website.com`

Since Inferno was intended for actual human beings, more effort was made to create a ''modern'' web browser, unlike its counterparts in Plan 9, `charon` has rudimentary JavaScript support. Nevertheless this was done a very long time ago, and the browser is now unmaintained and fairly broken. Using it will frequently crash the system. It is in my opinion considerably worse then the Plan 9 browsers, which at least fails with less noise.

The mail client and server in Plan 9 is called `upas`. Setting up a working mail server requires a bit of work, but it's not too bad, the 9front FQA† gives helpful hints in section 7.7. Once configured, the command line `mail`, and the `acme Mail` clients are easy enough to use, and can even interface with GMail. 9front also includes a handful of useful shell scripts, in addition to the aforementioned `wurl2txt`, you have scripts such as `ircrc`, `hget`, `hpost` and `webpaste`, which is an IRC client, a simplistic `wget` and `curl` clone, and a pastebin shortcut, respectively. Wait... an IRC client and a `curl` implementation as a *shell script?!?* Oh yes. You can do amazing stuff with Plan 9, here is an implementation of `telnet`:

```
#!/bin/rc

clonefile=/net/tcp/clone

if(! ~ $#* 2) {
        echo Usage: telnet ip port >[1=2]
        exit usage
}

<[4] $clonefile {
        netdir='{basename -d $clonefile} ^ / ^ '{cat /fd/4}
        echo connect $1|$2 >$netdir/ctl || exit 'cannot connect'
        cat $netdir/data &
        cat >$netdir/data
}
```

Note that the last examples are programs written in an attempt to make Plan 9 more compatible with external operating systems. Such support is fairly poor in classic Plan 9 and Inferno, for instance Plan 9 has an `ssh` client and NFS support, but only for the practically deprecated version 1 and 3 of these protocols. Only 9front supports the 2nd version of the `ssh` protocol. The reason for this lacking support, beyond

*) Update, 9front recently ported the NetSurf browser, which mitigates this problem somewhat.
†) That is not a typo, the 9front user guide is called ''Frequently Questioned Answers.''

code rot, is that the developers considered such external technologies as a bit daft. Plan 9, and Inferno, themselves are largely network and security agnostic, since these things are implemented in the filesystem. You don't connect to a remote Plan 9 machine with `ssh` or `ftp`, you just import its filesystem. As for chatting, you can easily write a peer to peer chat program using nothing but `echo` and `tail`.

Inferno has a few tools for communicating with external systems, such as telnet and mail programs, but it's doubtful that they will find any practical use today. It also includes a collaboration suite of programs, such as `chat` and `whiteboard`, but these programs can only talk to other inferno clients, which makes them about as useful as a Klingon-only peer to peer service.

At this point we may also pause and reflect on the futility of even trying to recreate a popular web browser in an alternative OS. The team who worked on `charon` were not merely experienced programmers, they had been developing operating systems and programming languages for three decades, and yet they failed miserably. `mothra` and `abaco` are smooth and elegant in comparison, but they are also literally 5000 times smaller then Firefox. I find that most people are surprisingly naive about the web. *''why haven't the programmers developed a decent browser?!?''*. Why indeed. How do you suppose the programmers could develop something akin to Firefox, when the source code for that browser is thirty times larger then the entire Inferno project, kernel, libraries and userland combined?

Not coincidentally the modern web flies totally in the face of the UNIX philosophy. Why can't you `grep` for webpages using regex? Why can't you `cat` a webpage, or nicely format it with `fmt`? Why can't you customize webpages using shell scripts, or edit them using your favorite editor? Why doesn't your browser come with a manual? How wonderfully productive the web could have been, but alas, it's an overblown entertainment system of filth.

3.3. Office

9front			**9ferno**		
name	src	man	name	src	man
troff	9279	2	cook	1770	1
(prep)	17286	11.5	brutus	3873	1.5
spell	1475	1	ebook	4367	1
dict	7322	1.5			
lp	1028	3			

There is only one ‘‘office suite’’ in Plan 9, good ’ol `troff`. `troff` is easily one of the most underrated programs on the planet. It is extremely rare nowadays to see anything but manpages written in it (and even that is sadly scarce), but the document preprocessor is capable of producing professional documents in PDF or HTML. Many of the reference books mentioned in this article are written in `troff`, and in fact the article you are reading now has been written in `troff` on a Plan 9 machine. Thanks to unicode support in the Plan 9 version of this tool, you can easily write documents with exotic characters as well, something that isn’t easily done in Tex or DocBook.

The biggest hurdle to overcome when learning to use `troff`, is simply that it’s a markup language. So the code must be written first, and then you can compile it to PDF or HTML. But this is easier then it sounds, the manual for `ms`, providing generic article macros to `troff` is only 3 pages long. The syntax is also easy: write a macro or `troff` command, such as `.SH` for a section header, `.LP` for a left-adjusted paragraph, or `.PP` for an indented paragraph, on a line by itself, then add the contents verbatim after it. That’s it. This syntax fits perfectly with the UNIX tools, for example, to print out the raw text from an `ms` document: `sed '/^\./d' doc.ms | fmt`, or the section headers and their line numbers: `grep -n '^.SH' doc.ms`. In contrast, LibreOffice’s ODT format uses unreadable XML. Below a short letter is written in `troff`. The `troff` code constitutes 5% of the text. In comparison, the same letter in Markdown would contain 7% syntax code, LaTeX 34%, HTML 83% (minimum), DocBook 122%, and lastly, ODT 15,824% (no, really!). Only Markdown is even close to competing with `troff` in terms of ease of use, but it certainly cannot compete with its power.

Example letter in `troff`:

```
.SH
Hello grandma!
.LP
Hi, just wanted to ask how you are doing?
How’s the knitting progressing?
I’m still using the sweater you made me last winter,
it’s keeping me warm and cozy :)
.PP
Sent to you by my Plan 9 box
```

Result:

Hello grandma!

Hi, just wanted to ask how you are doing? How’s the knitting progressing? I’m still using the sweater you made me last winter, it’s keeping me warm and cozy :)

Sent to you by my Plan 9 box

Another tricky problem in `troff` is that much of the functionality is spread across multiple commands. For example, if you want tables or math equations in your document, you need to run the document through the `tbl` and `eqn` preprocessors first. Luckily, Plan 9 comes with a handy script that automates this for you. Running `doctype doc.ms` will print the commands required to convert the document to postscript. So to read the document in `page`, all you need to do is: `doctype doc.ms | rc | page` or you can export it to PDF: `doctype doc.ms | rc | ps2pdf > doc.pdf`. As mentioned `troff` is a seriously underrated program. It is nearly a 1000 times smaller then `texlive`, but for casual usage it gives you basically the same functionality. With `troff` you can convert a thousand page book to PDF long before LibreOffice has even finished its loading screen.

`troff` has not yet been ported to Harvey, but it is available in Plan9Port. There are subtle differences though, for instance you need to tell Ghostscript where it can find the Plan9Port specific fonts. The easiest way to do this is with `psfonts`. To demonstrate: `doctype doc.ms | rc | tr2post | psfonts > doc.ps; ps2pdf doc.ps; xpdf doc.pdf`. The unicode aware Plan9Port `troff` is actually a very nice and sane replacement for GNU `groff`.

As for Inferno you can read manpages locally with `man -f`, but it does not include `troff` itself or any other macros then `man`, so it isn’t practical for general documentation. As mentioned Inferno does come with an SGML (aka. DocBook) editor called `brutus`, it also includes a tool called `cook` that can

convert SGML to HTML or LaTeX. The output formats are hopelessly outdated however (using HTML 3 for instance), and the editor is somewhat fidgety. Nevertheless it does work, and it's somewhat surprising that UNIX has no such equivalent. Inferno also include tools for reading and writing EPUB's, but these too are very outdated. Sadly Inferno has been neglected for a long time. But in its time, it was a very nice complement to Plan 9, and with the recent 9ferno and Purgatorio forks, it may perhaps thrive again.

Printing

There are no printer commands in Inferno, which makes sense, since it's much more practical to just use whatever printer command the host system provides. There are no spell checkers or dictionary tools either, although the tools needed to create such programs are certainly there. (see discussion below)

Plan 9's printer command is `lp` (equivalent to `lpr` in UNIX). Driver support for physical printers are very limited, but you can print output as a postscript file: `lp -dstdout file > file.ps`. And if you have a UNIX machine nearby hooked up to a printer, you can always just do: `lp -dstdout file | ssh unixpc lpr`

Spell checking

Plan 9 includes a spell checker called `spell` and a dictionary program called `dict`, similar to WordNet in UNIX. (to use it, some 3rd party resources must first be installed - see `/lib/dict`.) As for the old Writers Workbench, aka. `diction`, there is sadly no such thing in Plan 9. The spell checker also has limitations, it only supports English, and worse, it uses binary dictionaries without support for local wordlists. You can however write your own custom spell checker easily enough:

```
deroff * | tr A-Z a-z | tr -c a-z '
´ | sort | uniq | comm -13 /lib/words -
```

This is especially handy for non-English languages. For example, to spell check my Norwegian documents, I first import the `aspell` dictionary as plain text from a UNIX machine: `ssh unixpc | 'aspell -d nb dump master | aspell -l nb expand' | tcs -f 8859-4 | sort > /lib/words.no`. With this in place some simple Norwegian spell checking functions might be:

```
fn lower{ tr A-ZÆØÅ a-zæøå }
fn words{ tr -c 'a-zæøåA-ZÆØÅ''' '
´ | tr -d '''' | sort | uniq }
fn nospell{
        dict=/lib/words.no
        if(test -f words.no) dict=words.no
        for(word in '{deroff $* | lower | words | comm -13 $dict})
                if(! grep -s '^'$word'$' $dict) echo $word
}
```

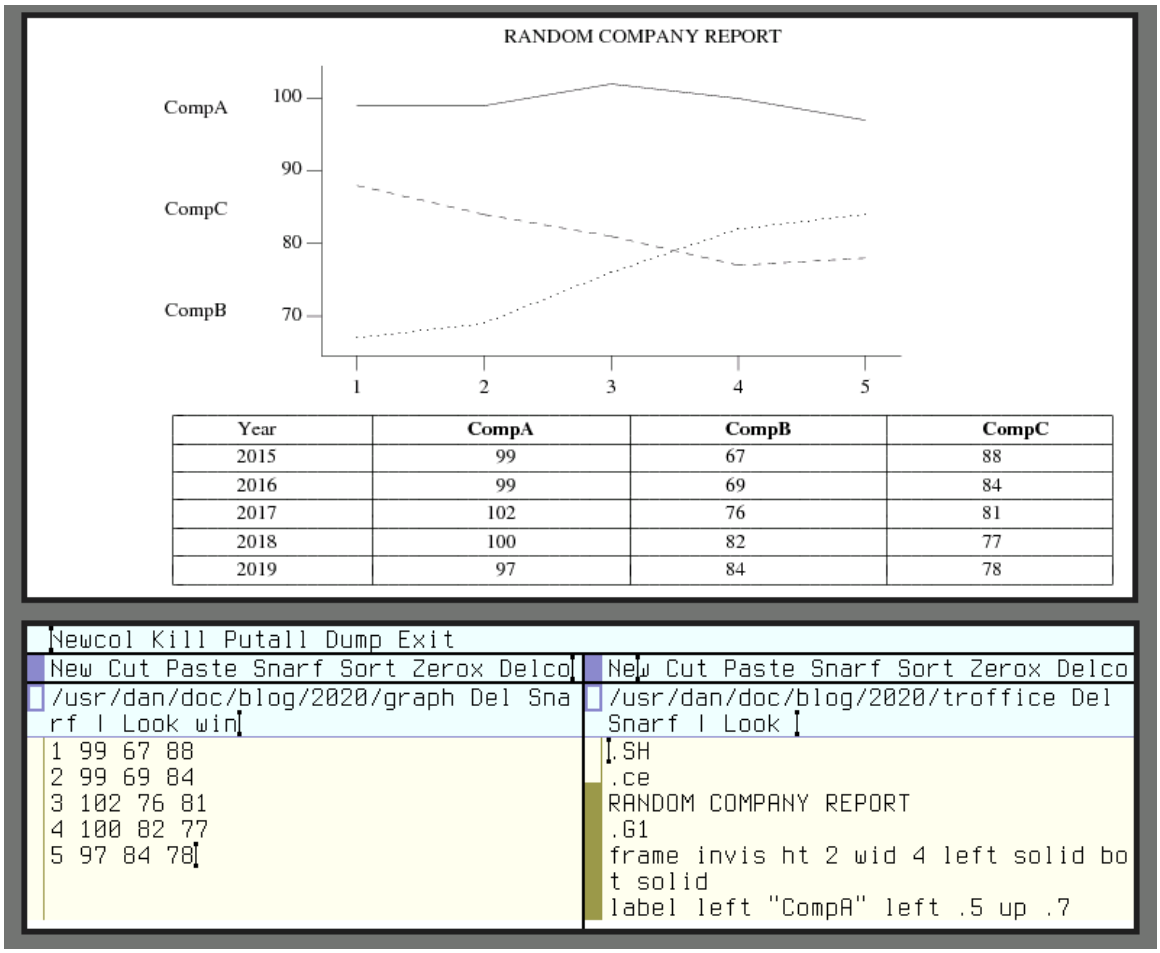
The above functions are fairly self explanatory (unfortunately `grep` is necessary here since `comm` will not handle non-English characters well). The `nospell` command will detect the presence of local dictionaries and use them. You can easily write a script that scans your documents and auto-generates such a local dictionary based on the master dictionary. Beyond optimization, it is sometimes useful to have project specific wordlists. For example, almost 15% of the unique words in the document you are reading right now, are words that I don't want to add to my master dictionary, but that I nevertheless want to check. This includes words such as ''xvesa'', ''wiley'', ''vax'', ''thompson'', etc.

Of course, our script is very basic.* It has no understanding of grammar or word prefixes, and it will only print a list of presumable misspellings, with no clue as to how they are meant to be spelled. But with a sufficiently large master dictionary, the script works surprisingly well, even on par with the LibreOffice spell checker. Beyond flexibility, our solution has the added benefit of training our brains to spell correctly, as we need to manually fix our own spelling mistakes.

---

*) *Classic Shell Scripting* from O'Reilly, pages 331-349, demonstrates how to write a more intelligent spell checker in awk, if you are interested.

Spreadsheets, accounting, databases, graphs, etc...



Year	CompA	CompB	CompC
2015	99	67	88
2016	99	69	84
2017	102	76	81
2018	100	82	77
2019	97	84	78

TrOffice

So you can write documents in Plan 9. Big Woop. What about spreadsheets, graphs, databases and the like? Well, have you heard of `awk`? Suppose you have exported your personal accounting in LibreOffice Calc to CSV:

```
Date,Expenses,Category,Comments
2020-07-30,1000,rent,i hate paying rent...
2020-08-01,24.50,food,grocery store
2020-08-02,1.35,travel,buss ticket
```

To find out how much money you spent in August, or just how much you spent on food in August, you can:

```
awk -F, 'NR>1 { sum+=$2 } END { print sum }' account.csv
awk -F, '$1=/2020-08/ && $3=/food/ { sum+=$2 }
         END { print sum }' account.csv
```

Creating a personal account script, or even a general purpose CSV editor, would not be hard. But wouldn't it be nice to actually *see* these values in a nice graphical table? As long as you don't need an interactive table, this is easy too:

```
cat << end > account.tbl
        .TS
        expand center allbox;
        c c c c
        l n l l
        .
        end
sed 's/,/       /g' account.csv >> account.tbl # whitespace here is a tab
echo .TE >> account.tbl
tbl account.tbl | troff | page
```

If you want a nice graph of your food expenses in the first 8 months of 2020, you can run these commands:

```
echo .G1 > graph
echo draw solid >> graph
for(month is '{seq 8}){
        awk -F, '$1=/2020-0'$month'/ && $3=/food/ { sum += $2 }
                END { print sum }
        ' account.cvs >> graph
}
echo .G2 >> graph
grap graph | pic | troff | page
```

See the `tbl`(1) and `grap`(1) manpages to understand what is going on (they are only 3 pages each). The point here, is that even though Plan 9 has no interactive office tools, you can nevertheless accomplish advanced office tasks with relative ease. On my own Plan 9 box I have written a small script that can take any generic CSV file and produce a graphical `tbl` table. The most difficult part of this is figuring out what kind of content each column should have, and to break up a long table into smaller chunks, since `tbl` will not handle tables that overflow the page well. With these caveats in mind, the script is still only 40 lines of code. `troff` can do other things as well, for example, check out the `eqn`(1), `pic`(1) and `mpictures`(6), to see how you can add math equations, pictographics and pictures to your office documents. What about databases? On page 109 of *The Awk Programming Language* by Aho, Weinberger and Kernighan (hence the name a-w-k) they demonstrate how you can build a general purpose SQL database with 50 lines of `awk`. Of course, you can also just write your databases in `ndb`(8).

3.4. Shell

9front			**9ferno**		
name	src	man	name	src	man
rc	5321	8.5	sh	8534	17
ape/sh	15,560	-	mash	7263	13
			tiny	533	4

For compatibility reasons, Plan 9 has a `pdksh` like shell in `ape/sh`, but the systems default shell is `rc` (run command). `rc` was written from scratch to replace the UNIX shell. The biggest difference between the two is that `rc` uses a C like syntax, and that it represents all arguments as lists, making array operations seamless. All in all the Plan 9 shell is very elegant, it has added vital functionality missing in the old Bourne shell, while removing all the warts. It does not however support any math operations or interactive features, such as job control, history substitutions or line editing. These features are provided by external tools. For example the window manager provides a text-editor-like window for the shell, making substitutions unnecessary, just edit and copy-paste as you would in a normal text editor. It also provides a full copy of your shell session in `/dev/text`, so printing your command history is simple: `grep '^;' /dev/text` (assuming that your prompt is '; '). The window manager provides job control, and `hoc` (or `bc`) takes care of mathematics.

Here are some other fun examples: The `""` command (no, that's not a typo, double quotes have no special meaning in Plan 9) in 9front is a 9 line shell script mimicking `!!` in UNIX (rerun previous command). Adjusting this script to mimic the `!$` or `^^` commands would be easy enough. As for

`clear`, `script` and `xclip` you can do all that with:

```
cat > /dev/text
cat /dev/text > typescript
cat text > /dev/snarf
```

The clipboard in Plan 9 is called the ‘‘snarf’’ buffer. However brilliant the UNIX developers were, they did have a quirky sense of humor (the name of this operating system is derived from Ed Woods legendary B horror movie *Plan 9 from Outer Space*, and I am not even going to tell you about their mascot...).

Inferno’s `sh` shell, not to be confused with the ‘‘sh’’ Bourne shell, is very similar to `rc`, but like the operating system itself, is more modular. You can for instance integrate `sh` with the graphical Tk toolkit and manipulate desktop applications, making it somewhat reminiscent of Tcl/Tk. There are other modules for math and string manipulation etc, in fact the shell has an FFI to Limbo, meaning that any compiled module or library in Inferno can be loaded directly from its shell. Now that is *truly* next generation scripting! `mash` is the old version of this shell, whereas `tiny` is a stripped down shell, created for working on memory constrained systems. That consideration is a bit humorous when the ‘‘bloated’’ `sh` shell is half the size of `dash`!

Example of Shell Syntax

pdksh

```
echo ${a}string
rm *.{mp3,ogg}
alias user="echo $USER"
end(){
  shift && echo $*
}

set -A a hi guy\!
echo $((${#a[@]}+1))

for w in ${a[@]}; do
  echo -n "$w "
done

if [ ${#a[@]} = 0 ]
then exit 1
else
  v=${a[1]}
  echo bye ${v%!}
fi
echo $?

while true; do
  (subproc)
done

case "$1" in
  (Abe|Bob) echo Hi
    ;;
  Carl) echo Hey
    ;;
  *) kill -s KILL $$
     ;;
esac
```

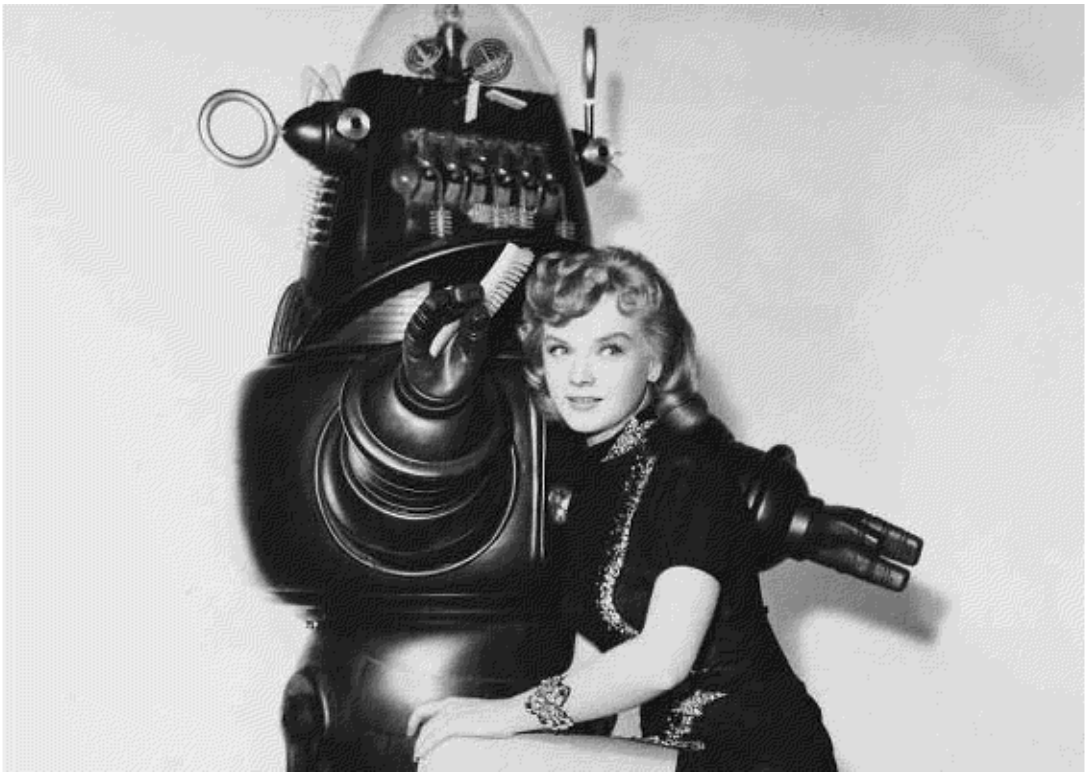
rc

```
echo $"a^string
rm *.^(mp3 ogg)
fn user{echo $user}
fn end{
  shift && echo $*
}

a=(hi guy!)
echo $#a + 1 | hoc

for (w in $a) {
  echo -n ’’$w’ ’
}

if(~ $#a 0){
  exit empty
} if not {
  echo bye $a(2) |
  sed ’s/!//’
}
echo $status

while(){
  @{subproc}
}

switch($1){
case Abe Bob
  echo Hi
case Carl
  echo Hey
case *
  echo kill>/proc/$pid/ctl
}
```

inferno-sh

```
echo $"a^string
rm *.^(mp3 ogg)
fn user{cat /dev/user}
fn end{
  echo ${tl $*}
}

a=(hi guy!)
echo $#a + 1 | calc

for w in $a {
  echo -n ’’$w’ ’
}

if{~ $#a 0}{
  exit empty
}{
  echo bye ${index 2 $a} |
  sed ’s/!//’
}
echo $status

while{}{
  @{subproc}
}


if {~ $1 Abe Bob}{
  echo Hi
}{~ $1 Carl }{
  echo Hey
}{
  kill ${pid}
}
```

3.5. Applications



	9front		9ferno	
name	src	man	src	man
echo	28	0.5	33	0.5
cat	29	1	42	0.5
ls	290	2	289	2
cp	178	1	214	0.5
wc	315	0.5	271	0.5
sed	1258	4	809	-
awk	6275	5		
tar	1229	2		
sort	1506	2.5	118	0.5
tail	325	1	340	1
ps	149	1.5	51	1
file	1500	1		
grep	1520	1.5	135	1
date	57	1	63	4.5
hoc	1897	1.5		
calc			2383	3.5
p	81	0.5	127	0.5

You will find many of the expected UNIX commands in Plan 9 and Inferno, but beware that they behave differently and unpatriotically (few flags), so keep an open mind. `hoc` and `calc` are similar to `bc`, while `p` is similar to `more` in UNIX. As you can see, a few common UNIX utilities are not provided with Inferno, but the `os` command lets Inferno run applications from the host operating system it's running on. To add `awk` to your Inferno system for instance, just write this shell script:

```
#!/dis/sh
if { ~ $#* 1 } { file = /fd/0 } { file = $2 }
os -d $emuroot^'{pwd} awk $1 $file
```

The `/sys/src/cmd` directory in 9front contains about 160 utilities (not including subdirectories). These have a combined source code of 65,000 lines and manuals numbering 140 pages. 9legacy and Harvey has 130 utilities, 50,000 lines of code and 130 pages of manuals. Plan9Port has about 80 utilities, 26,000 lines of code and 230 manual pages. The `/appl/cmd` directory in Inferno holds about 150 utilities, with 46,000 lines of source code and 140 manual pages.

The source code here is about four times smaller then BusyBox, although the functionality is much greater. What makes Inferno and Plan 9 tools especially powerful, isn't the tools themselves, but how the operating system connects them together. Networks are presented as plain files for instance, so `cp` also does the job as `scp` in UNIX. Remote chat clients can be written with `echo` and `cat`, window manager utilities too for that matter.

Lets compare some of these numbers to their Linux counterparts. GNU `sed` and `grep` are about 150 times larger then the Plan 9 versions, but they do pretty much the same job. The Plan 9 versions are in my opinion slightly better since they have proper handling of unicode, and use a standard `egrep` like regex library. `ps` is 30 times bigger. (to be fair the BSD versions of `grep` are about the same size, `sed` twice as big and `ps` ''only'' 10 times larger) Plan 9 presents process information in `/proc` , and its version of `ps` just reads these files. Linux adopted the `proc` filesystem from Plan 9, but few utilities actually use it, worse still, the implementation presents a significant security risk since these files can be seen by all programs (which is why the BSD's don't use it by default). In contrast each process in Plan 9 has its own private view of the filesystem, so it's easy to shield parts of the system from processes if you need to. This example illustrates why the developers felt that UNIX needed to be rethought. You cannot ''fix'' UNIX simply by adding Plan 9 technologies to it, it just adds to the mess. To fix UNIX, you need to *remove* stuff!

More Applications

9front			**9ferno**		
name	src	man	name	src	man
zuke	1689	2	wm/mpeg	4731	2
play	99	0.5	wm/wmplay	158	-
page	1630	2	wm/view	402	0.5
paint	746	1	wm/dir	459	-
games/snes	4336	2	wm/tetris	730	0.5
games/mines	3177	1	wm/snake	334	-
games/sudoku	932	1	wm/sweeper	274	~0

There are only a few GUI applications in Plan 9, such as the document and image viewer `page`, the music player `juke`,* the infamous `catclock` and a handful of simple games. Much of the traditional desktop functions are provided by `acme` and the shell, for example, `acme` can be used as a file manager and an email client, and shell tools such as `date` and `bc` provide a clock and a calculator. When it comes to 3rd party applications, Plan 9 doesn't have a package repository per se. A community repository of about 100 contributors were provided by Bell Labs. The contributions were in source form and required some assembly to install. The server has since closed down, but the contents are still accessible from `https://9p.io`. A lot of this old software will not compile on 9front though. This fork provide their own, smaller but not totally insignificant, set of extra packages on their web site.

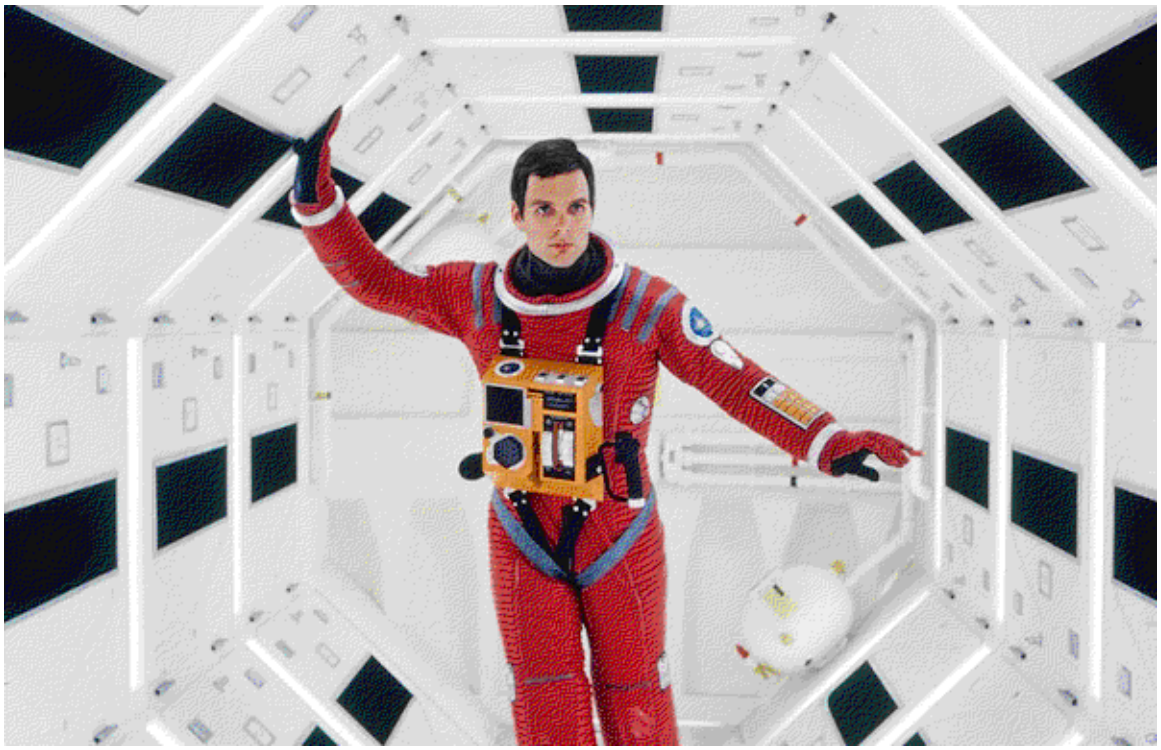
As for Inferno, it has a more traditional desktop feel and ships with more GUI tools. `mpeg`, `avi` and `wmplay` are used to play videos and music, but the tools are old and unmaintained, and limited audio drivers make them virtually impossible to use. An image viewer, file manager and several other utilities are

*) 9front uses zuke and play instead.

also provided. The documentation talks about 3rd party applications, and the developers clearly wanted to facilitate such development. But the project failed commercially and no repository of additional packages were ever made. Some developers, such as MJL, have released extra software for Inferno on their private github repositories, but that is rare.

The absence of a large repository of precompiled packages may shock UNIX (well, Ubuntu) users, but keep in mind that these operating systems are *not* UNIX. Backwards compatibility was not a goal, and many practical necessities that UNIX developers take for granted, are not provided in Plan 9, such as links and user ID numbers, not to mention X. Porting applications from UNIX to Plan 9 is possible, but non-trivial. In any case, if you really did port all the UNIX software to Plan 9, fleshing out all the libs and patching the kernel to Kingdom come, the system would become an inconsistent and bloated behemoth, exactly like UNIX. Plan 9 is different *for a reason!*

3.6. Desktop



Plan 9			**Inferno**		
name	src	man	name	src	man
rio	5382	6	wm	609	2.5

Plan 9 and Inferno each have their own window manager. `rio` is very minimalistic and is reminiscent of `twm` in appearance. But it has some unique features not found anywhere in UNIX. Although it is interactively used solely by mouse action, it presents a number of plain text control files. You can thus easily create window manager shell scripts to handle such things as auto startup or window tiling for example.

New windows in `rio` are always terminals, when executing a GUI program, the terminal window morphs into this application. In this way `rio` windows naturally gain a pseudo-tilling ability. You just create the initial windows where you want them, and then run whatever programs you wish in them, they won't normally be resized or moved. No popup windows exist in Plan 9 either to distract you or block your view. If you want to make your setup permanent, run `wloc` and use that as a basis for editing the `rio`

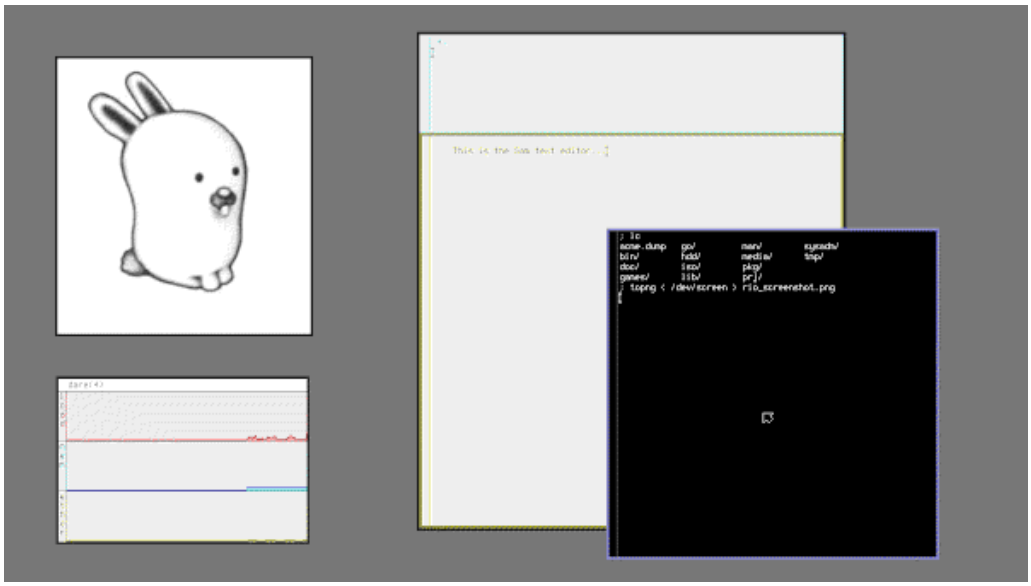
startup file `$home/bin/rc/riostart`. Supposing you want an `acme` window taking up the left half, and a terminal window taking up the right half of a 1920x1080 screen at startup, your `riostart` would look something like this:

```
#!/bin/rc
window -r 0 0 960 1080 acme
window -r 960 0 1920 1080 rc
```

Of course this is just a shell script, so you can create however many `rio` startup files as you want, one for every type of configuration. `rio` can also run inside a `rio` window, thus providing you with an easy workspace mechanism. Using the terminal in Plan 9 is a requirement, and many common tasks are done with basic shell commands such as `echo` or `cat`, here are a few examples:

```
echo resize -r 0 0 1600 900 > /dev/wctl # maximize window on 1600x900 screen
echo master 100 100 > /dev/volume       # set speaker volume to maximum
cat /dev/window > sshot_window          # take screenshot of window
topng < /dev/screen > sshot_screen.png  # take full screenshot as PNG
sleep 600; echo BUGME > '#c/cons'       # send notification in 10 minutes
```

Plan 9 doesn't have any graphical toolkit, instead it provides a low level graphics library. The desktop was really designed with the purpose of running terminals. In addition `acme`, arguably Plan 9's main user application, was designed to do everything. Consequently very little attention is given to traditional GUI designs. No doubt new users will find this alienating, but it lies at the very heart of the systems simplicity and elegance. The source code in `/sys/src/libdraw` totals 6000 lines of code, and cover just 14 pages in the manual.



Plan 9 Desktop

The most interesting thing about Plan 9's humble window manager is that it is controlled by writing text strings to a set of files. This makes it very easy to write window manager scripts. To give an example; `rio` does not have the ability to automatically tile windows, but its fairly trivial to create such functionality:

```
#!/bin/rc
# tile - tile windows
# usage: tile

# gather some information
screensize=(0 0 ‘{echo $vgasize | awk -Fx ’{print $1, $2}’})
windows=‘{for (win in /mnt/wsys/wsys/*)
        if(dd -if $win/wctl -bs 128 -count 1 -quiet 1|grep -s visible)
                echo ‘{basename $win}
}
fn left{ awk ’{printf("%s %s %d %d %d %d",$1,$2,   0,   0,$5/2,$6 )}’}
fn right{awk ’{printf("%s %s %d %d %d %d",$1,$2,$5/2,’$b’,$5, ’$e’)}’}

# auto tile windows
if(~ $#windows 1)
    echo resize -r $screensize > /mnt/wsys/wsys/$windows/wctl
if not {
    echo current > /mnt/wsys/wsys/$windows(1)^/wctl
    echo resize -r $screensize | left > /mnt/wsys/wsys/$windows(1)^/wctl
    windows=‘{echo $windows | sed ’s/^[^ ]+ //’}    # shift
    step=‘{ echo $screensize(4) / $#windows | bc }
    b=0; e=$step                                    # begin, end length
    for(i in $windows){
        echo current > /mnt/wsys/wsys/$i/wctl
        echo resize -r $screensize | right > /mnt/wsys/wsys/$i/wctl
        b=‘{ echo $b + $step | bc }
        e=‘{ echo $e + $step | bc }
    }
}
```

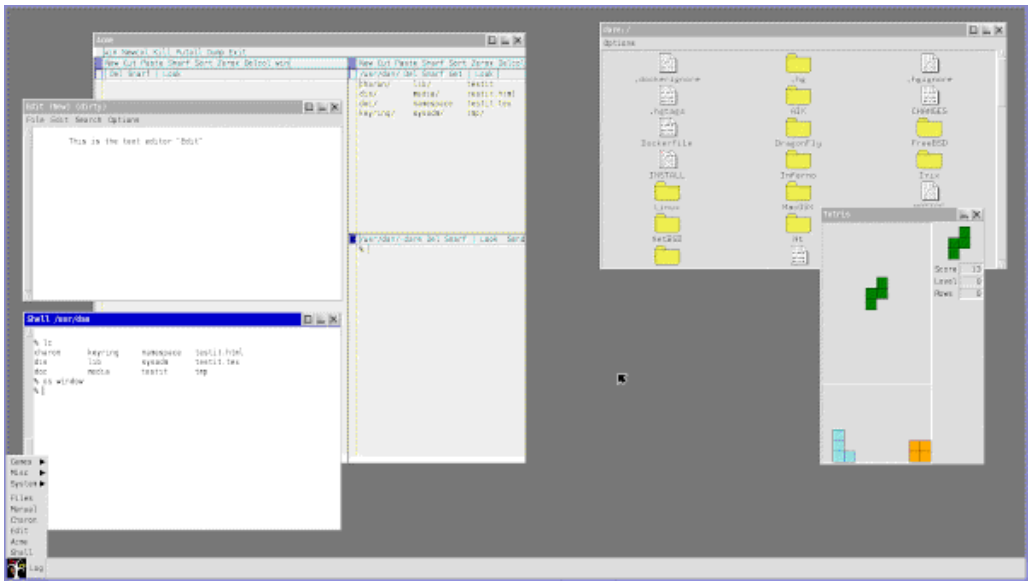
The `tile`, script just does a bunch of `awk` arithmetic to figure out the coordinates for the left half of the screen, which it gives to the first window, and then it carves the right half of the screen up into however many chunks are needed for the remaining windows. One complication to beware of is that the window in question must be made the currently active one before it is resized, since `rio` will not allow you to resize an inactive window. You can write all kinds of window manager extensions to suit your needs, on my Plan 9 box for instance, I have a 20 line shell script that provides virtual workspaces.

Harvey includes a variant of `rio` called `jay`, which has a simple panel and window buttons. It doesn’t really add anything except a thin layer of recognizability for newbies. As for Plan9Port it does include a window manager called `rio` which tries hard to look like the Plan 9 equivalent. But because of limitations in UNIX, it does not present its control interface as files, neither can it run nested sessions inside itself, nor does it morph new programs into the same window. So in reality it is little more then a fake look-a-like, and frankly there are far much better alternatives, such as the Plan 9 inspired `wmii` or `dwm` tiling window managers.

Finally, the Inferno desktop is somewhat of a hybrid between the Plan 9 ideas and traditional GUI systems. `acme` is included, pretty much everything is presented as files and the shell in very similar to that of Plan 9. Like `rio`, `wm` can also run as a window inside itself. But desktop applications use a clone of Tk, which carries much more of the GUI burden, and gives a recognizable feel. Applications are launched from a startup menu, and GUI programs run in a terminal will popup as new windows. `wm` does present plain text control files like `rio`, but working with them requires that you learn a bit of Limbo and Tk programming first. This is both good and bad, on the one hand creating traditional looking desktop applications is much easier in Inferno, since you don’t need to invent a toolkit first. And if you have experience with Tk, you will be making graphical apps in no time. On the other hand, like traditional desktops, the implementation is inflexible. It is not possible to extend the window manager with simple shell scripts like we did for Plan 9 for instance.

The window manager is actually `wm/wm` (which also starts `wm/toolbar`). The top level wm directory holds about 50 other desktop utilities, including a few simple games. The total amount of source code lines for all of these are 41,000 lines, and their manpages total 56 pages. The GUI depends on the

code in `/libdraw`, `/libprefab` and `/libtk`, which totals 30,000 lines of code, and cover the 9th section of the manual, which spans 120 pages.



Inferno Desktop

Comparing the source code here to UNIX counterparts is almost laughable. For instance, the default X window manager `twm`, is 30 times larger then the Inferno desktop, even though it is much more frugal. As for the X Window System it is 140 times larger then all the GUI applications and support libraries in Inferno combined.

Plan 9 as a Daily Driver?

But one may ask; sure Plan 9 and Inferno are interesting and novel, but are they actually *useful?* Would any sane person really use them as a daily driver? No.* To quote from section 0.1.3 - ‘‘Plan 9 is not for you’’, in the 9front FQA:

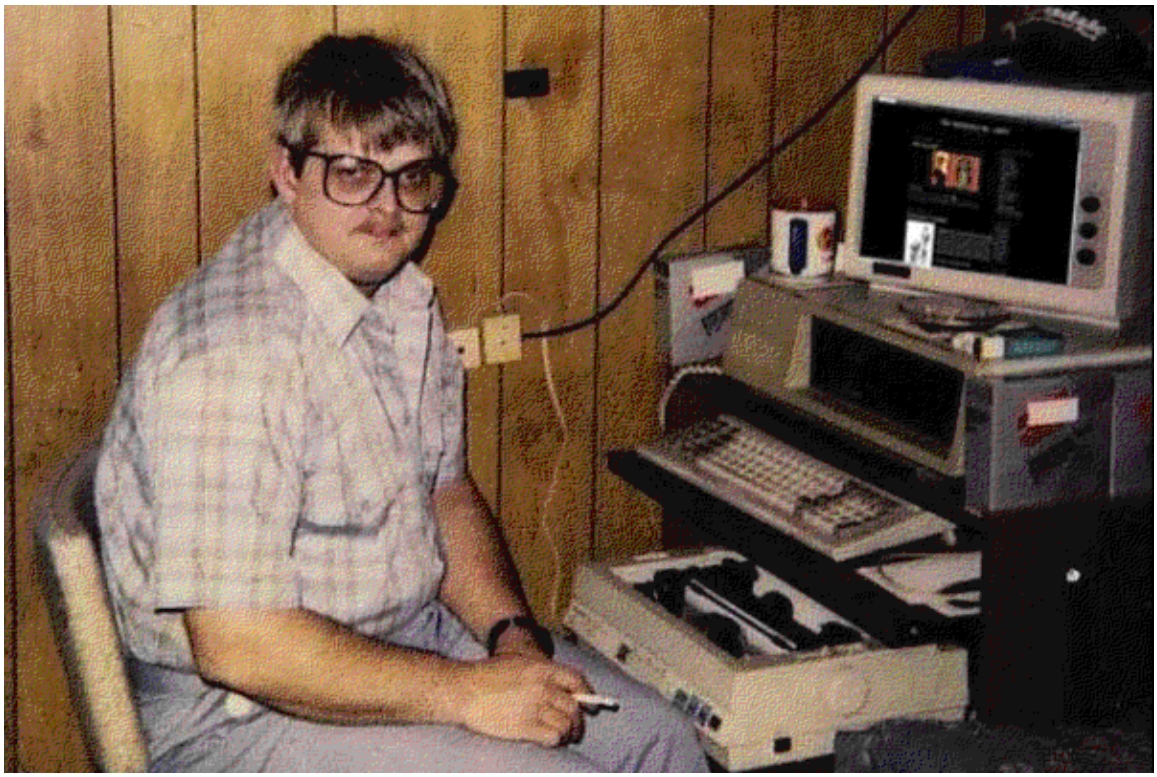
> *Let’s be perfectly honest. Many features that today’s ‘‘computer experts’’ consider to be essential to computing (JavaScript, CSS, HTML5, etc.) either did not exist when Plan 9 was abandoned, or were purposely left out of the operating system. You might find this to be an unacceptable obstacle to adopting Plan 9 into your daily workflow. If you cannot imagine a use for a computer that does not involve a web browser, Plan 9 may not be for you.*

Rob Pike described Plan 9 as ‘‘an argument.’’ And in many ways, that’s what it is. It demonstrates that it is possible to design a modern operating system, with graphics, networks and multiprocessing, while still holding on to the UNIX design principals. It provides a truly marvelous and unique experience that is fun to play with, and it gives a hint at what computing could have been like. But it really is a system *from outer space*, and is not at all suited for this world. So much the worse for humanity.

PS: If you cannot be persuaded from using Plan 9 as a day to day desktop, go with 9front. Not only is it actively developed, but it includes many features that you will likely need (and expect), such as video playback and wireless networking. In contrast to classic Plan 9, it is nearly useful.

*) Then again, sanity is overrated.

3.7. Programming



The Plan 9 C compiler collection in `/sys/src/cmd/cc` includes 11,000 lines of code, and section 2 of the manual, covering the system library, spans 390 pages. There isn't a single C compiler in Plan 9, instead each supported architecture has its own. This makes cross-compilation very easy, just use the `6c` compiler to compile amd64 binaries on an i386 system, instead of the i386 compiler `8c`. To compile and install an entire 64-bit Plan 9 system on a 32-bit system, just run `cd /sys/src ; objtype=amd64 mk install` (this only takes a few minutes).

Plan 9 uses its own C dialect, which is discussed in `/sys/doc/comp.ps` , see also Francisco J Ballesteros book *Introduction to OS Abstractions Using Plan 9 From Bell Labs.* The biggest difference between this dialect and ANSI C, is that Plan 9 doesn't have insane headers, preprocessors or `ifdef`'s. `exits` return a string instead of an int, and system libraries actually provide decent string, unicode and multiprocessing support. Of course since everything is a file in Plan 9, C programs are significantly cleaner. There are no sockets, links or ioctls to grapple with. All the source code for Plan 9 is included in the installation, and the `src` command will automatically open a programs source code in your editor. Use this command for what it's worth, it is a gateway to profound beauty and wisdom!

A meager POSIX compatibility layer is also provided with `ape`, such as an ANSI C compiler in `ape/cc`. But don't get your hopes up, most UNIX software will not compile in Plan 9. See `/sys/doc/ape.ps` for more information. Perl, Python and Go have also been ported to Plan 9.

Inferno uses the Limbo programming language exclusively. The sources in `/limbo` count 24,000 lines of code, and section 2 of the manual, covering the system library (you don't use syscalls in Inferno instead you use Limbo modules), spans 400 pages. Sadly some of the documentation for Limbo is outdated, but a good starting point can nevertheless be found in `/doc/descent/descent.pdf`. See also the book *Inferno Programming with Limbo*, which can be freely obtained from the internet. If you like Go, then you will love Limbo! It's similar in its design principals, but simpler with saner syntax.

Go is really a reimplementation and expansion of Limbo for UNIX, in comparison it is 80 times bigger. And there is some movement in the Harvey and Plan9Port projects to rewrite these in Go. As for C, the compiler collection from GNU is 800 times bigger compared to that of Plan 9.

3.8. Kernel

The kernel source for Plan 9 is in `/sys/src/9` , (9legacy and Harvey also include a 64-bit fork of this in `/sys/src/9k` , which holds about 80,000 lines worth of code), and contains about 250,000 lines of code. Section 9 of the manual, discussing kernel features, cover 23 pages. The Inferno kernel is in `/os` , it contains about 200,000 lines worth of code, and the 10th section of the manual, which spans 120 pages, discusses both the hosted and native aspects of the operating system. The code can be divided into these five categories:

	9front	**9legacy**	**Harvey**	**9ferno**	
name	src	src	src	src	comment
boot	24	11,782	12,326	13,978	startup procedures
ip	16,150	14,764	15,087	16,235	network
port	50,642	43,781	44,700	47,034	portable (main) kernel facilities
pc	94,291	103,797	105,875	72,468	x86 (32/64) drivers
bcm	12,706	19,272	19,572	n/a	arm (32/64) drivers
other	55,128	85,509	88,590	55,178	other drivers and things
sum	228,941	278,905	286,150	204,893	in total

As you can see, apart from a different startup procedure in 9front, the overall complexity of these Plan 9 variants, and even 9ferno, are fairly equal. This similarity is deceptive however, as there are significant technical differences between 9front and 9legacy, let alone Plan 9 and Inferno. Harvey though is very much in sync with 9legacy, whereas its sister project, JehanneOS follows 9front more closely.

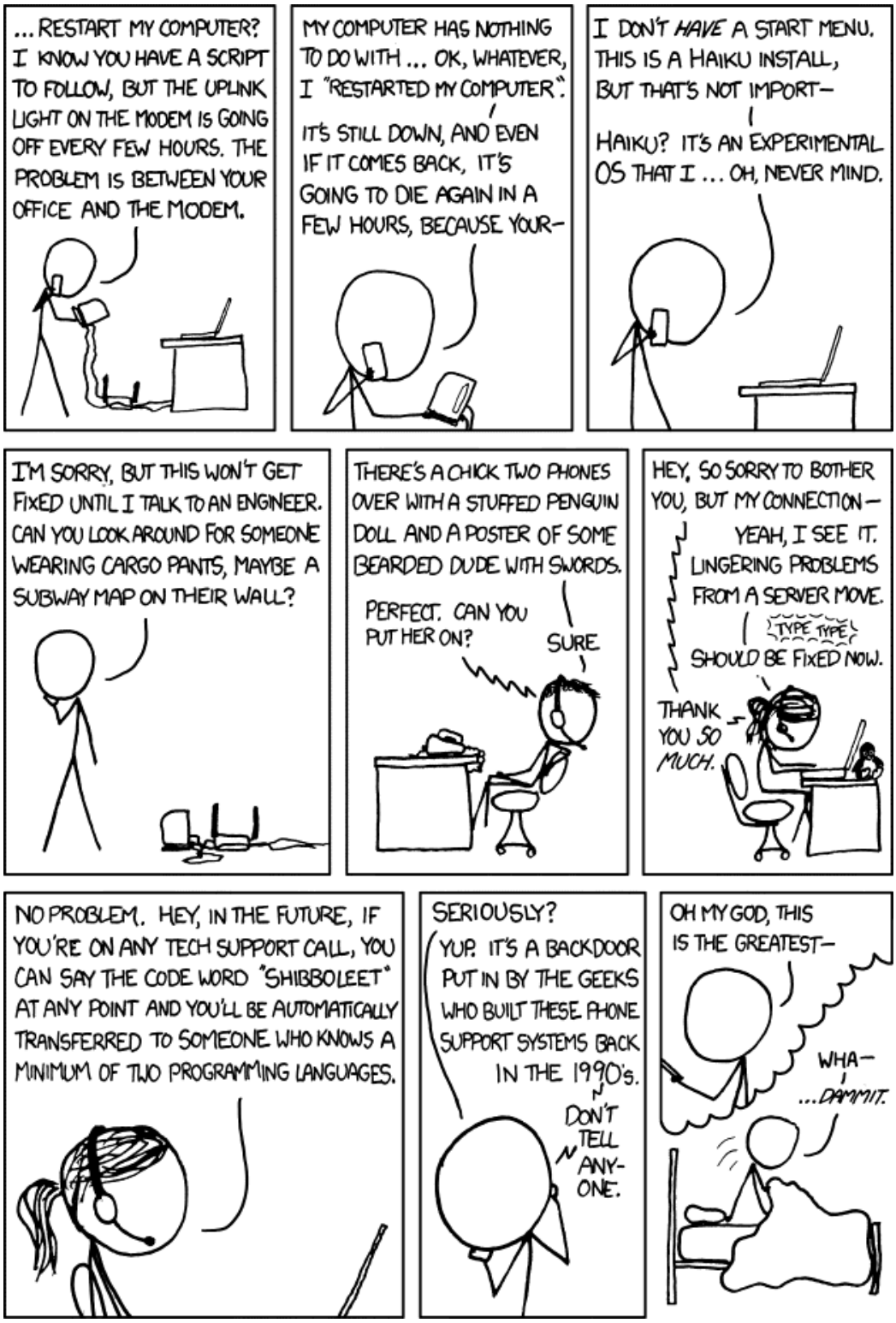
Cross platforming was a major goal for the developers, and both the Plan 9 and Inferno kernels support a wide array of architectures. Of course many of these are obsolete now, for example, both of these systems can be compiled on Irix, but amd64 (aka. ‘‘64-bit’’) support has only appeared recently. For a more in depth look at the Plan 9 kernel see Francisco J Ballesteros book *Notes on the Plan 9 3rd Edition Kernel.* The Plan 9 and Inferno kernels are about a hundred times smaller then the Linux kernel, about the same size as Minoca or Minix, and about twice the size of ancient UNIX kernels like 4.3BSD.

Capabilities

With the small kernel size mentioned above, it is worth pausing a bit and reflecting on what these kernels can actually do. First of all, the filesystem in Plan 9/Inferno runs on a network protocol (9P). Whether the file in question is actually on the local computer, or somewhere on the web, is irrelevant. Naturally a solid security framework is built deep into the system to ensure that all such transactions are safe, again whether or not such transactions happen locally or via the internet, is irrelevant, they are secured regardless (Plan 9 does not have `root` or `setuid` problems, neither does it implicitly trust foreign kernels, like UNIX does). What this means, is that Plan 9/Inferno is network and security *agnostic.* Any program running on these systems gets these things for free.

In addition, each process in Plan 9/Inferno has its own private view of the filesystem, or ‘‘namespace.’’ So in effect, all processes run inside their own mini-jails. It is easy to control just how much, or how little, access each process should have to the system. But this technique was not devised primarily to *isolate* resources, but to *distribute* them. For example, if you want to run a Mips binary on a PC, just import the CPU from a Mips machine. If you want to debug a system that crashes during startup, just import its `/proc` on a working machine, and run the debugger from there, etc. Since all resources are files, and all processes have their own private view of files, and networks and security are transparent, you can freely mix and mash any resources on the net as you see fit. In fact, Plan 9 was *designed* to run as a single operating system, spread out across multiple physical machines on a network. No other operating system, that I am aware of, is even close to providing such capabilities. In recent years modern UNIX systems have begun incorporating unicode, jails and snapshots, technologies that Plan 9 had invented in the early 90’s, but their implementations have been clunky, clumsy and laborious to learn in comparison.

4. MINOCA, SerenityOS and HAIKU



In this chapter we will take a closer look at what is often referred to as ‘‘alternative operating systems’’, that is systems other then Linux, BSD or UNIX™.* There are great many to choose from; ReactOS, FreeDOS, IcarOS and ArcaOS are popular reimplementations/continuations of the proprietary operating systems Microsoft Windows, MS-DOS, Amiga OS and OS/2. However nostalgic, these systems have nothing in common with UNIX, and are therefore too dissimilar to warrant further analysis. SkyOS and Syllable are opensource and UNIX-like desktop systems that do fit well with our theme, but they have been dead now for so long that I felt it best to leave them be. The history buff in me would have loved to study Multics and OpenGenera, but they too, aside from a purely historical connection, have nothing in common with UNIX.† And there are a plethora of smaller educational, special purpose or just plain weird operating systems, such as Oberon, Visopsys, MikeOS, FreeRTOS, KolibriOS, MenuetOS, DexOS, TempleOS, and so on and so forth. I find these systems very interesting, but again, they are too dissimilar to UNIX to fit nicely within our discussion. So, in the end I have opted to analyze just three, out of the great many alternative operating systems out there; Minoca, SerenityOS and Haiku. These are all fairly mature and practical, opensource and UNIX-like, and are thus a natural fit.

Minoca and SerenityOS are very new projects, the former is the brain child of two young developers who formerly worked at Microsoft. Their motivation for writing an operating system from scratch was the realization that all other major OS’s are several decades old, and consequently weren’t very well designed for modern hardware and use cases, specifically by not being sufficiently modular. In this sense, it bears some resemblance to MINIX, in terms of goals at least. They designed a very small and elegant UNIX-like operating system within just five years. Minoca is targeted for embedded systems, and thus have a much smaller goal then most of the other projects discussed in this paper. Nevertheless, it is a feature rich and POSIX compatible command line environment, well suited for embedded applications or OS studies.

SerenityOS and Haiku are more general purpose however, and ostensibly, they have many similarities; They are both developed by former Apple employees, they are both UNIX-like retro-looking desktop systems, and they are both snappy and practical. But there are significant differences too. For one SerenityOS is newer and smaller, and lacks both maturity and features in comparison to Haiku. In fact, it has yet to be released as an installable ISO. Typically, its users build it straight from source from a more mainstream OS, say Linux, and run it inside a virtual machine. Yet, the young project has many fascinating tricks up its sleeve; From JSON kernel internals and a lite Rust-like programming language called Jakt, to monthly developer updates on Youtube and community support on Discord. Despite the retro aesthetic, SerenityOS feels very much like a modern re-take on UNIX.

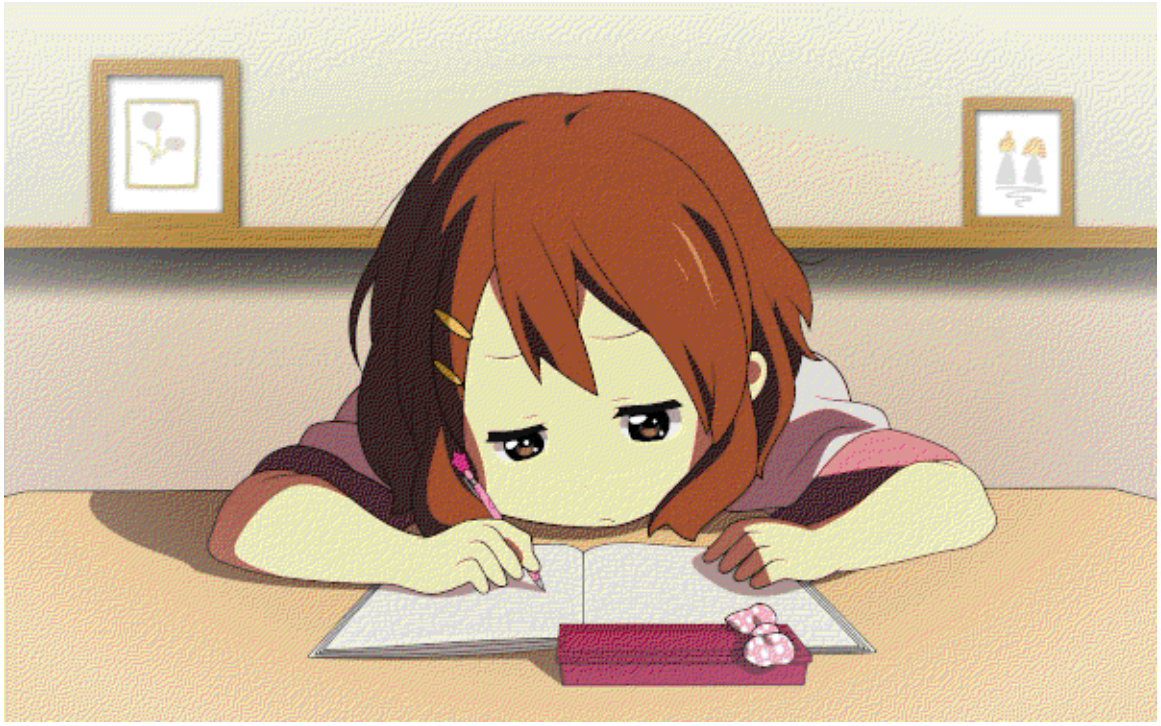
Haiku is a much older project, its roots goes back to the early 90’s, when a former French Apple developer started his own rival company called Be Inc. Their multimedia oriented operating system, called BeOS, used aggressive multi-threading and a clean design without backwards compatibility cruft, to produce a blazingly fast desktop. While a good POSIX compatibility layer allowed the system to easily import UNIX applications. Despite having a good product that was seriously considered as a base for Mac OS X, the company failed and was eventually sold to Palm Inc in 2001. Before its demise many of the BeOS key technologies were released as opensource, but it took many years before the open Haiku fork reached a usable level. The first alpha came in 2009, the first beta in 2018. Yet, despite being still in beta, the retro-looking OS is insanely fast and surprisingly feature rich. With a 3rd party package repository that isn’t all that shabby compared to more mainstream competitors.

---

*) Of course Plan 9 and Inferno discussed in the previous chapter would definitely fall into the category of "alternative operating systems", but I felt they were interesting enough to deserve a chapter of their own

†) To be clear; From a surface level, Multics is indeed reminiscent of UNIX Version 1, but already by Version 4, the two had diverged beyond recognition. So, appart from historic fascination, a comparison of the two will be irrelevant here.

4.1. Text Editors



SerenityOS		**Haiku**	
name	src	name	src
TextEditor	971	StyledEdit	3142
		Pe	56,810

Minoca ships with `nano`, and includes `vim` and Emacs in its package repository.

SerenityOS has a few UNIX command line editors in its Ports collection, namely `ed`, `vim`, and `nano`. But there is only one text editor by default, the appropriately named TextEditor. Superficially, it looks like Notepad with syntax highlighting for a handful of programming languages. But in true Serenity fashion, it has some surprising tricks up its sleeve, such as rendering Markdown and HTML source files in real time!

With that in mind, the above statistic is quite absurd. Programs in SerenityOS do indeed have a shockingly low source code line count, and there are several reasons why, beyond good programming that is; For one, software in SerenityOS is very new, lacking in essential features and fixes. We should expect these numbers to swell quite a bit before the system reaches a mature level. Perhaps more importantly, programs have dependencies. 971 lines of C++ is not enough to get such a graphical intensive text editor up and running. The five source files making up TextEditor include 57 library files, which in turn include 284 library files, which in turn include even more library files, which in turn include more or less the entire OS. Of course, this also holds true for any other program. A simple text editor in Linux may depend upon GTK, which depends upon X11, which depends upon the Linux Kernel, which depends upon GCC, and so on... It is easy to count lines of code in a directory, but working out what the code actually does, and what it depends upon is another matter entirely. This general warning aside though, SerenityOS do tend to use libraries much more aggressively then most. This is by design of course, and it is very impressive that the default libraries allow you to write such a functional GUI program in less then a 1000 lines of code!

In addition to `nano`, Haiku has two graphical editors, the Notepad like StyledEdit and the slightly more advanced Pe (Programming Editor). Pe is more or less a simplified Gedit, with syntax highlighting support for a few dozen programming languages. It is quite nice for such a small project, and the HaikuDepot package manager also has `vim`, Emacs and Kate.

4.2. Internet

SerenityOS		**Haiku**	
name	src	name	src
Browser	3250	WebPositive	10,205
Mail	750	Mail	13,502

The situation in Minoca is quite frugal. It includes `wget` by default, and you can install `curl` and `ssh` but there are no web browsers or mail clients in its repository. Of course you *do* have a C compiler in the repo, so if you want a web browser, go make one!

SerenityOS has its own web browser, and as with everything else in this operating system, it is written from the ground up, with no 3rd party components. With that fact in mind, it can render an impressive amount of web sites out there, including not so shabby JavaScript support, but it obviously falls short in comparison to mainstream alternatives. The Mail client is minuscule, and will almost certainly crash if you try to use it with a real Email account. The Ports collection includes some classic UNIX web tools, such as `wget`, `curl` and `openssh`. SerenityOS also ships with its own Web server.

For a small OS Haiku's native web browser is surprisingly good, but don't let the low source code count deceive you, most of the work is being done by the multi-million line WebKit backend. But even this huge backend isn't nearly big enough to compete with Firefox or Chrome, which translates into the fact that many webpages and services simply won't work. `wget` and `curl` are included as well, and happily `lynx` and a few other browsers, that do a far worse job then WebPositive, can be installed through the package manager. It also has a Mail client and a web, FTP and SSH server.

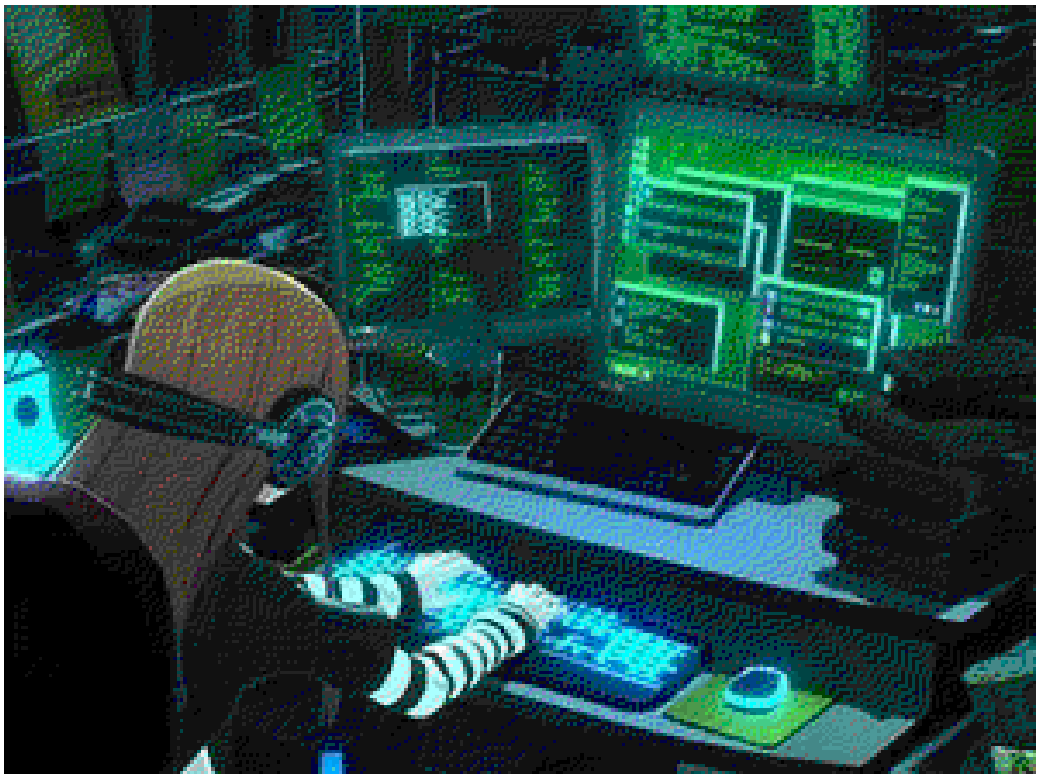
4.3. Office

You can install `groff`. in Minoca. What more do you need?

SerenityOS does not include any of the common productivity or office applications that a Linux user might expect, instead it relies heavily upon Markdown for all its documentation needs. All the manpages, and all the other documentation is written in plain old Markdown. For a new operating system this makes a lot of sense! Everything is HTML these days, so just stick to that. Why on Earth would you go to great length recreating the 50 year old troff, or the 7 million lined LaTex? Of course HTML is terrible, but writing it in Markdown will take away a lot of the pain at least, and it is relatively easy to convert simple Markdown to troff, LaTex, DocBook, or any other format you may need. As mentioned, Serenitys TextEditor also renders Markdown and HTML in real time, making it a very productive office tool if you constrain your documents to those formats. You will also find a PDF reader, that actually renders some PDF documents without crashing, on occasion, and a fairly powerful Spreadsheet application using a custom JavaScript dialect.

Haiku ships with `groff` (GNU troff) and a PDF reader by default, but that's not all. The repository includes LibreOffice, Calligra and Scribus, and these killerapps works like a charm! There is DocBook and LaTeX as well, in fact you will find a great many of the popular opensource offerings available for Haiku.

4.4. Shell



Minoca		**SerenityOS**	
name	src	name	src
sh	17,933	Shell	12,756
chalk	26,749		

Minoca includes its own Bourne compatible shell, and a unique scripting language called `chalk` (naturally it's undocumented - but `apps/ck/README.md` in the Minoca source tree will give you some pointers at least), while Haiku uses `bash` as its default shell. Both operating systems include several other alternatives in their package repositories, Haiku even includes the Plan 9 inspired `es` shell!

SerenityOS has its own shell, which looks very much like a UNIX shell, but with some differences. Like the Plan 9 shell, lists and arrays are seamless, you can do pairwise concatenations (or "Juxtapositions"), ReadWrite redirections, functions with explicit arguments, and more! In short, the Serenity Shell, behaves like a UNIX shell *should* have done. The Ports collection also contain some bona fide UNIX shells if you need them, such as `bash`, `dash`, `oksh` (OpenBSD Korn Shell) and `zsh`.

Example of SerenityOS Shell Syntax (see page 34 for comparison)

```
echo $a(string)
rm $(glob {mp3,ogg})
alias user=whoami
end(first){
    echo $first
    shift && echo $*
}

a=(hi guy!)
expr ${length $a} + 1
```

```
for w in $a {
    echo -n "$w "
}
if [ ${length $a} = 0 ] {
    exit 1
} else {
    v=${remove_suffix ! $a[1]}
    echo bye $v
}
echo $?
```

```
loop {
    {subproc}&
}

match $1 {
    Abe|Bob { echo Hi }
    Carl { echo Hey }
    * { kill -KILL $$ }
}
```

4.5. Applications

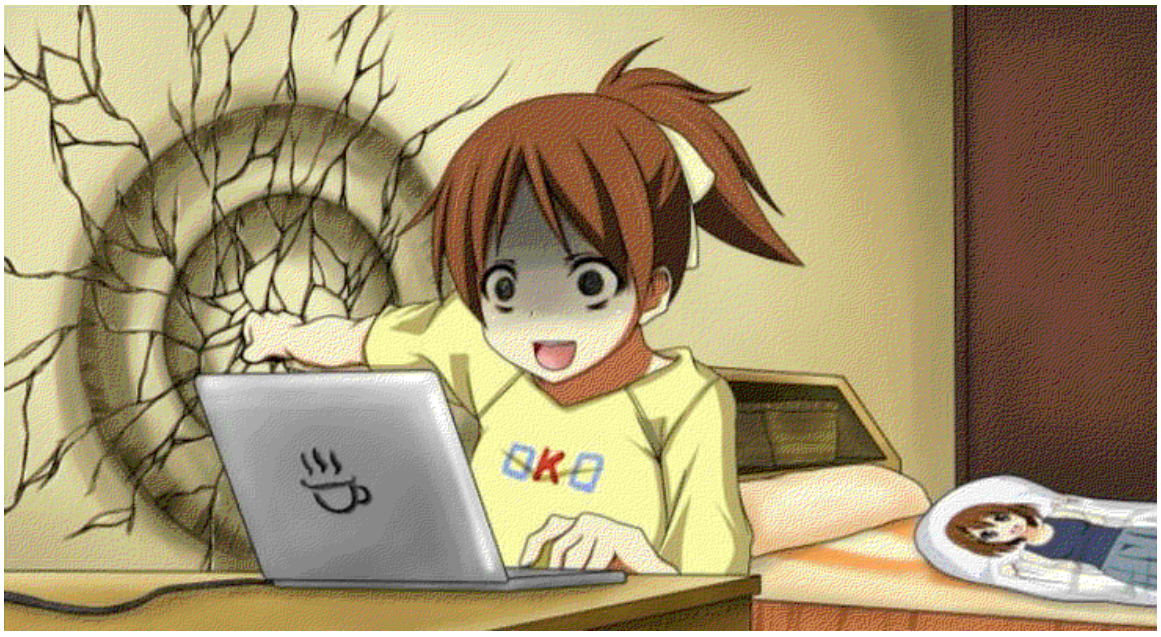
Minoca		**SerenityOS**				**Haiku**	
name	src	name	src	name	src	name	src
echo	130	echo	111	PDFViewer	850	BePDF	124,868
cat	377	cat	52	Spreadsheet	5317	MediaPlayer	22,087
ls	1606	ls	472	Calendar	319	ShowImage	6617
find	1509	find	493	SoundPlayer	1397	Sudoku	2760
cp	200	cp	80	ImageViewer	547		
wc	309	wc	97	Hearts	1358		
sed	3393			Chess	963		
		tar	219	Minesweeper	783		
sort	1141	sort	34				
tail	289	tail	68				
ps	1769	ps	162				
		file	152				
grep	954	grep	264				
date	283	date	57				
		less	502				

Minoca is almost exclusively devoted to the shell, their collection of standard utilities is distributed in a single binary called `swiss`. The approach is similar to BusyBox in Linux, but the source code is three times smaller. The entire suit contains some 80 utilities, and have a source code count of 67,600 lines. GNU tools that don't have an equivalent in this collection, such as `awk`, `tar`, `file` and `less` can be installed through the package manager (but there's no `bc` or other calculator which is a curious oversight).

SerenityOS also develops its own UNIX-like command line tools, it sports some 180 utilities, which have a baffling low combined source code count of 22,800 lines. And again, GNU tools that don't have a native alternative, such as `awk`, `sed` and `bc` can be installed from Ports.

Technically, Haiku has about 150 native shell utilities, with a combined source code of 105,000 lines. None of these tools follow in any way UNIX design principals however, and they are about as useful as the Windows CMD utilities. But Haiku also imports the GNU coreutils and other basic UNIX tools, which gives it a rich and familiar command line environment. This power is somewhat of a happy accident, as the developers seems totally distracted by the desktop, and just slapped on a large pile of GNU, because they couldn't be bothered to do it themselves.

4.6. Desktop

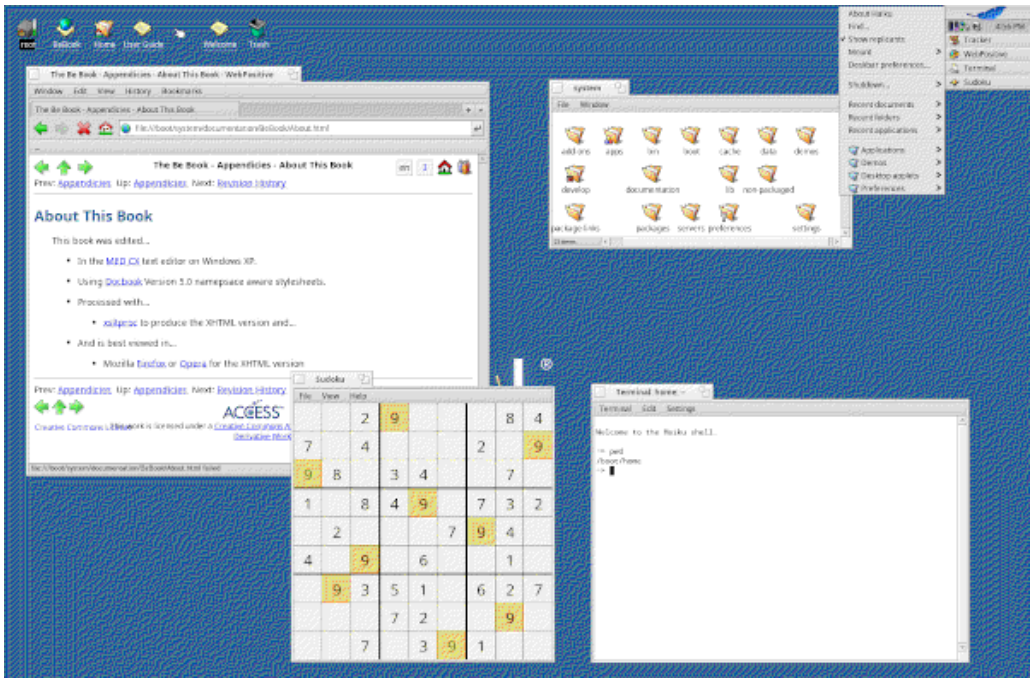


Both Haiku and SerenityOS are very GUI oriented operating systems, so much so that it's actually very difficult to find anything specific about their desktops, since bits and pieces of them are sprinkled throughout their respective source codes. In this respect these operating systems are somewhat reminiscent of Windows, where there is no clear distinction between the core system and the graphical desktop. Unlike Windows however both Haiku and SerenityOS are lean mean super machines, by the time Ubuntu shuffles into the office, they have been up for hours and are feverishly sipping their fourth cup of coffee.

At first impression, the Haiku desktop may seem quite spartan and novel. You'll appreciate the responsiveness early on though, and with usage, you'll also notice that there is much careful though behind this deceptively simple GUI. Some examples: The window title only goes part way across the top of a window border. This is not some failed attempt at fancy aesthetics, as one might first suspect, it has a purpose; Hold down the Windows button as you drag one window title onto another, and the two windows will be tabbed together. Another example: Some applications, like Workspaces, DeskCalc, have a tiny yellow arrow at the bottom right corner of the window. Hold down the Windows button and click and drag this arrow to place a clone of this application as an applet on the desktop. Both examples illustrate good GUI design: Don't add tabs to individual applications, and create some applet which duplicates already existing functionality; let the desktop take care of such interactivity, and all applications will benefit. A final example: Left click the simple load bar on the start menu, navigate to "Threads and CPU usage", and you can view how much CPU usage each segment of each program is using in real time! The multi-threaded nature of Haiku is really very impressive, you could run many movies simultaneously without lag even back in the 90's. Something even modern computer systems will struggle with today.

The Haiku desktop sports about 60 graphical utilities in total, which have a combined source code of 320,000 lines. For a fully fledged desktop these numbers are quite frugal, but the Haiku applications are surprisingly feature rich. In fact just by their appearance alone you would have expected their source code to be 10-20 times larger then they are. ShowImage and MediaPlayer can handle just about any image, audio and video format, and as already mentioned a fairly decent PDF reader, web browser and programmer friendly text editor are also available. And there are some nifty little extras, such as Workspaces and

Sudoku. There aren't nearly as many 3rd party applications for Haiku as there are for Linux or BSD, but HaikuDepot still ships with thousands of additional packages available for download, including dozens of games, and even bigwig applications, like Blender, Qt_Creator and Slack. The only painful omissions here are Firefox and Chrome, but it is understandable that porting such huge behemoths would be difficult.

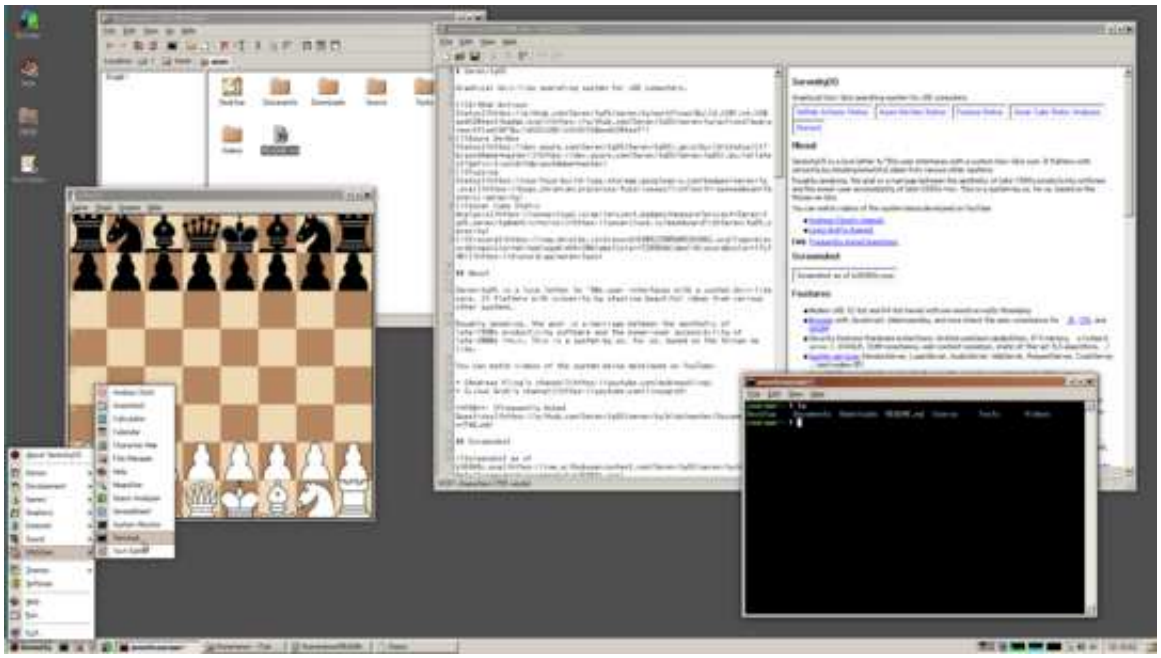


Haiku Desktop

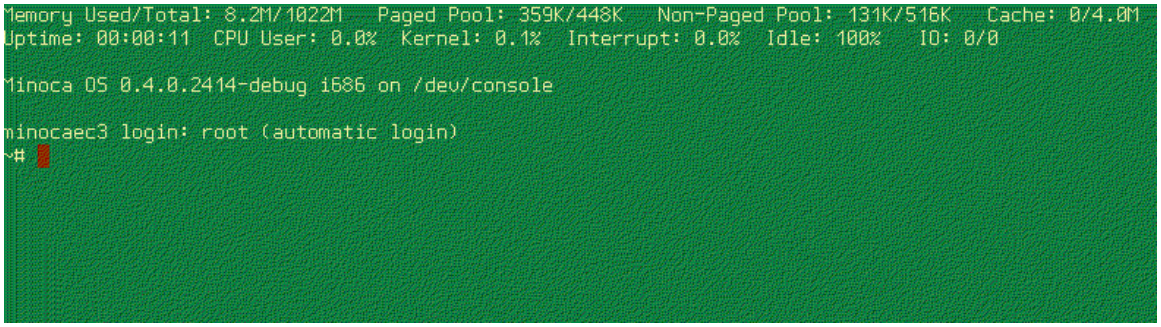
By comparison, Serenity has a much more familiar GUI. On the surface level it seems like a Windows 95 clone. But this system too is deceptive; Once you pop up the hood and take a look inside, you will notice that *everything* is different! We have already seen some revealing statistics. The source code of Serenity applications are so simple and elegant that it is breath taking! They are on par with Plan 9 or Inferno, and that is saying a lot. The baffling thing, for me at least, is that Serenity embraces many technologies that I personally view as evils when it comes to simplicity; JavaScript, C++, OOP, Windows 95, GUI, Youtube and Discord, are some keywords that comes to mind. Yet, they somehow manage to produce a stunningly slim but powerful operating system. I find this as surprising as it is delightful, and it suggest a view contrary to my own: Perhaps designing a simple OS is not so much about what abstract philosophy you follow, but simply the act of rewriting it from scratch under the wise management of a small development team. What has probably contributed most to Serenitys simplicity, is its strict rule of only shipping in-house code. There are also other examples that support this view: Inferno and KolibriOS also provide GUI-rich desktops with very simple code. These three operating systems have very different internals, and very different design philosophies, but they are all written from scratch by a small team of developers.

The Serenity desktop includes over 90 graphical utilities, with a combined source code of 80,000 lines. In terms of applications this desktop is surprisingly rich, including 10 games, and most of the usual suspects, such as a web browser, email client, audio player, calendar, PDF and image viewer, and some fairly good development applications. Having that said, SerenityOS is Alpha software, and you will quickly notice that these desktop applications are both frugal and crashy. As mentioned, Serenity has strong aversion about importing 3rd party code. But it does have a set of 3rd party Ports that you can optionally install. It is somewhat fiddly to work with, as you have to compile the ports on the host system, not natively, at the collection only contains some 250 entries. Nevertheless, you will find some important UNIX tools there, and a handful of fun games.

Don't be too quick to dismiss alternative operating systems just because they are obscure. As a rule, the really innovative ideas come from small systems. SerenityOS is hands down one of the most pleasant development environments I have ever seen, and Haiku can serve well as a smooth and functional retro desktop. Like Windows however, neither of them can run without a GUI, limiting their usefulness in the server or embedded market. In this capacity Minoca really shines. Although you can run X in it (eg. `opkg update; opkg install xinit; startx`), there is little point in doing so, since about the only thing you can do with it is run a bunch of `xterm`'s. But the system is useful as a development platform for small applications, and the tiny system serves very well as an educational tool for learning how operating systems work.



Serenity Desktop and Minoca



4.7. Programming

Minoca does not include any programming utilities by default, except for shell scripting, but many of the usual suspects can be installed with its package manager, such as `gcc`, `perl`, `python` and `nasm`. (eg. `opkg update; opkg install gcc`)

SerenityOS supports shell and JavaScripting out of the box, in addition to C++ programming. Work is also under way to replace the C++ code base with Jakt, a custom built programming language that bears some resemblance to Rust. Brian Cantrill, of Solaris fame, has suggested that operating systems should be rewritten in Rust. SerenityOS might just become one of the first desktop oriented OS written in a Rust-like language.* Naturally, you can also find many other classic programming tools in the Ports collection, such as, `gcc`, `llvm`, `lua`, `nasm`, `php`, `python`, `ruby`, and `tcl`.

As for Haiku, it ships with most common programming utilities out of the box, and it contains quite a few extras in its repository besides, including `tcl`, `ruby`, `lua`, `nodejs`, `rust`, `erlang` and rudimentary support for Haskell, Lisp and Java. Whatever your personal choice of poison may be, Haiku should have you covered.

It could also be added here that unlike all the other operating systems analyzed (except for Inferno), SerenityOS and Haiku aren't predominantly written in C. Half the systems source code in Haiku (especially the GUI aspects of it), and all of the source code in SerenityOS, is written in C++. In this respect too they bear a resemblance to Windows, but unlike the Microsoft Leviathan, they are actually nice and sane environment to program in, especially SerenityOS. The entire section 2 and 3, covering system calls and libraries in the SerenityOS manuals only cover about 60 pages (though incomplete documentation contributes mightily to the simplicity). Both Haiku and SerenityOS provide desktops with surprising speed and elegance, the fact that the GUI aesthetics lag some 30 years behind the times, may have something to do with the outstanding performance.

*) See also https://redox-os.org for another example

4.8. Kernel

	Minoca	SerenityOS	Haiku
boot	16,173	351	43,758
kernel	104,363	74,150	100,010
drivers	151,604	4017	43,706
total	272,140	78,518	187,474

The actual Kernel core of these operating systems are fairly equal in size, and roughly comparable to OpenBSD in this respect, but more complex then the Plan 9 or Minix kernels. The main difference in overall size boils down to device drivers; whereas Minoca has some drivers, Haiku has few, and SerenityOS none. Nevertheless, we are comparing apples and oranges here. Minoca is a tiny gem that fits snugly in your pocket, intended for embedded use. SerenityOS is a much more GUI intensive affair. In theory it is intended as a daily driver, but for now it is very much a system by developers for developers. Whereas Haiku is an older and considerably more robust project, providing quite a fair competitor to more mainstream alternatives.

Conclusions

As mentioned there are many alternative operating systems out there, but the three we have looked at here illustrate nicely common strengths and weaknesses. Haiku is the most ambitious of the three, and with 30 years of history, including solid corporate backing in the past, it is mature enough to provide a rich set of applications and documentation. The innovative and responsive desktop augments these features, and can provide a real unique and refreshing experience for a hobbyist user. Yet good driver support is just too much for the developers to handle. And this isn't the worst of it, even if a new user against all odds manages to run Haiku on his laptop, he will likely just shrug and think, *I have no use for a system without a web browser*.

For SerenityOS the situation is even worse. Although it's understandable that the project doesn't provide install ISO's or device drivers at this early stages of development, it is worrying that the project isn't self hosting yet. The longer the developers postpone taking the step into real world, the harder it will be to transition away from the developer-only environment. And the fact that is relies on a 3rd party 3rd party packaging system, may hint of a perpetual beta-state project. Of course, that may not be a big problem for the developers. SerenityOS has provided, and will likely continue to provide, a fun and engaging programming environment like none other!

In contrast to Haiku and SerenityOS, Minoca has a much more narrowly focused goal. Targeting the embedded market means that the developers can safely ignore the vast amount of work needed to provide a desktop on commodity hardware. And it is because of that narrow focus that they managed to write a system from scratch within a few years. Yet programming alone does not make a useful product, Minoca has virtually no documentation, and without it, it is doubtful that it will see any serious usage.*

Alternative operating systems are valuable because truly innovative ideas is only feasible to experiment with on small systems. Unfortunately, modern expectations make such work difficult. Many users will simply demand that Linux must be 100% compatible with Microsoft Windows, and that any alternative operating system, no matter what the developers goals are, must be able to support their online shopping habits. Even when developers try to meet such unreasonable demands, their creative and tireless efforts are usually met with indifference and ridicule. Rob Pike estimated that 90-95% of the work in Plan 9 was implementing external standards, and Theo De Raadt has estimated that two thirds of the development effort in OpenBSD are on drivers alone. With that amount of work needed just to get a system booting, it's a small wonder that most who dabble with OS research are quite content with making the next Hanna Montana Linux distro. Whereas Bell Labs were rewarded several Nobel prizes for their work on UNIX, virtually no one has heard of Minoca, even though it is significantly larger then UNIX was in the beginning. The UNIX revolution in the early 70's didn't just happen because the system was so good, but also because expectations were so low.

*) Update, there has been no development in Minoca the last five years, so the project seems dormant.

5. BSD and MINIX

Even before the University of Berkeley discontinued its BSD project in the early 90's, two open-source forks sprang up to continued its development, NetBSD was a community of hardcore experts who continued the work of porting BSD to many platforms. FreeBSD focused on the popular 386 PC, and tried to make the operating system more user-friendly. It has since grown to be the most popular BSD variant and is extensively used on servers, no doubt because it is so ''user-friendly''. NetBSD is still around, and porting itself to architectures nobody has ever heard of, but has otherwise fallen into obscurity.

In 1995 OpenBSD forked from NetBSD due to some tiff among the developers. Unlike the other BSD's which are governed by democratic committees, OpenBSD enjoys the tyrannic rule of its lead developer Theo De Raadt. Its developers have an incredible, if not fanatic, focus on simplicity and security. Whether or not this is the ''right'' thing to do is a good topic for a flame war. But lets just be diplomatic and say that OpenBSD is the only BSD variant that the developers themselves would actually run on their own laptops, and not just in a terminal window on their MacBooks.

Lastly DragonFly BSD and Minix are research operating systems. DragonFly is a fork of FreeBSD with the goal of creating a distributed system, where you can install a single instance onto multiple physical machines (in this respect it is somewhat reminiscent of Plan 9). It hasn't reached that goal yet, but at least it has developed a nifty ZFS like filesystem, called HAMMER. Minix is a highly reliable self-repairing micro-kernel operating system, created by Andrew S. Tanenbaum. The kernel itself is a rewrite of UNIX V7 and has no direct connection to BSD (Linux started as a fork of Minix by the way, which according to Tanenbaum, took the wrong turn). But in later years the Minix developers have been working on importing the NetBSD userland and ports, which makes the practical system look and behave very much like this flavor of BSD.*

*) Update, there has been no development in Minix the last five years, so the project seems dormant.

Statistics for individual utilities are practically identical between DragonFly BSD and FreeBSD, and between Minix and NetBSD, therefore I have not included statistics for these two research operating systems in most of the following sections.

5.1. Text Editors

	OpenBSD		**NetBSD**		**FreeBSD**	
name	src	man	src	man	src	man
ed	2613	8	3071	9.5	2971	9.5
vi	25,023	24	52,365	21	31,694	24
mg	15,184	18				
ee					8644	8.5

`vi` here is of course the BSD variant `nvi`. `mg` is a simplistic reimplementation of Emacs, previously known as MicroGnuEmacs, that the OpenBSD developers have picked up from the gutter and nursed back to health. The FreeBSD editor `ee` (easy editor) is reminiscent of `nano`. No graphical editor is provided by default, but there is a plethora of choices in the Ports Collection.

The striking difference between the BSD's and Linux here, is that the BSD variants give you simple editors by default. `vim` is 30 times bigger then `nvi`, but for a casual programmer it doesn't really give you much more then syntax highlighting. You would think that this wasn't such a big deal, but what can I say, it's hard to see monochrome text when you're used to colors, just as it's hard to read a book without pictures if you're not used to it. Or to quote Rob Pike again: *Pretty printers mechanically produce pretty output that accentuates irrelevant detail in the program, which is* **as** *sensible* **as** *putting all the prepositions* **in** *English text* **in** *bold font.* `ee` is 20 times smaller then `nano`, while `mg` is 100 times smaller then Emacs! A major reason for this small size is that `mg` aims to be a *text editor* and doesn't include features like tetris or a psychotherapist. A good test for editor elegance is to read and edit its own source code, the BSD editors passes this test with flying colors!*

*) Well, not literally.

5.2. Office

	OpenBSD
	src
name	
mandoc	33,716

`mandoc` is an OpenBSD fork of `troff`, all the other BSD variants, as well as Illumos, use this as their standard `man` implementation. In overall complexity it is quite similar to the original `troff`, but the focus is different. `mandoc` doesn't support general purpose article macros, such as `ms` and `me`, and obscure preprocessors such as `pic` have been dropped. Instead the suite is exclusively devoted to producing technical manuals. It supports the old `man` macros, but the newer `mdoc` format, which gives much more fine-grained semantic control, is preferred.

For personal correspondence of course, you would need a different tool, and there are many to choose from in the Ports Collection. For example, if you have installed the `texlive` package, you can write a letter in LaTeX:

```
\documentclass{article}
\begin{document}
\subsection*{Hello grandpa!}
Hi, how are you my bearded progenitor?
Is the mainframe giving you trouble these days?
I just received the floppy disk you sent me,
looking forward to reading it,
once I have managed to locate my discarded disk drive :)

Sent to you by my BSD box
\end{document}
```

You can then compile this source to PDF by executing the commands `latex letter; dvipdf letter.dvi`. Alternatively you *could* write your correspondence in DocBook if you really had to. After installing the `dockbook` and `docbook-xsl` package you can write your DocBook letter (note that some of the details here will vary from system to system):

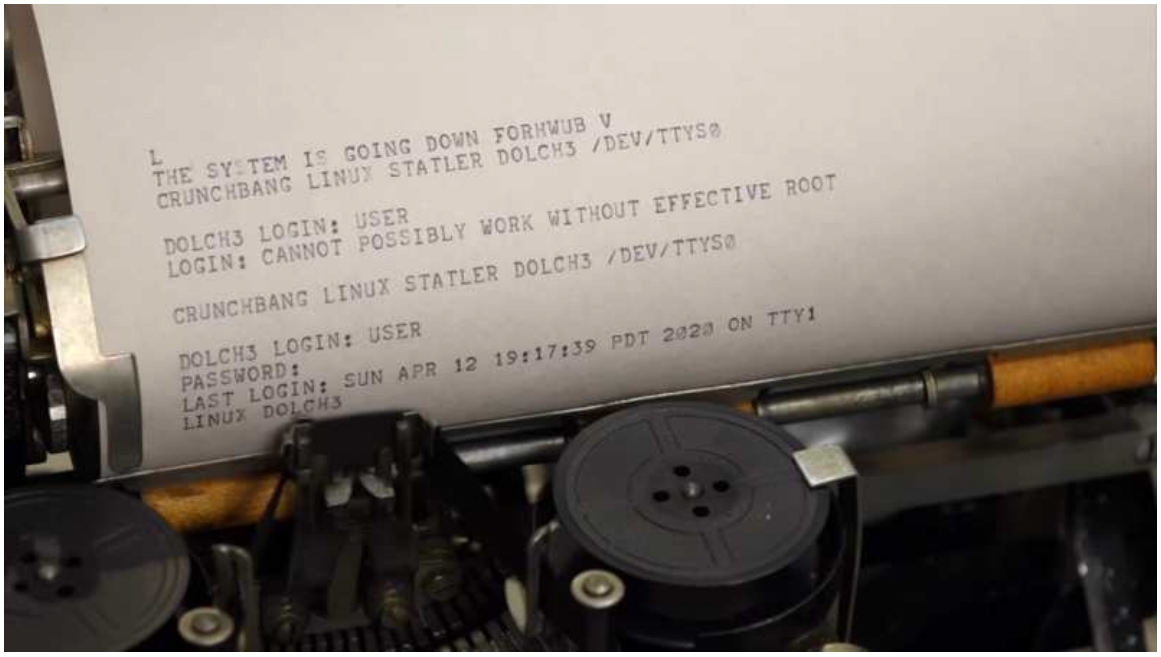
```
<!DOCTYPE article PUBLIC "-//OASIS/DTD DocBook XML V4.5//EN" \
  "/usr/local/share/xml/docbook/4.5/docbookx.dtd">
<article>
  <section>
    <title>Hello uncle!</title>
    <para>
    How's business my dear rich uncle in America?
    Is the stock market treating you OK these days?
    Thanks for the sneakers you sent me,
    I'm using them every day :)
    </para>
    <para>
    Sent to you by my BSD box
    </para>
  </section>
</article>
```

You can then compile this to HTML for instance with the following commands:

```
$ STYLEST=/usr/local/share/xsl/docbook/html/docbook.xsl
$ xsltproc -o letter.html $STYLEST letter.xml
```

Of course you can do all this in a comfy graphical office suite, such as LibreOffice. All of the BSD's have a large enough collection of ports to replicate virtually any Linux office environment, all of the office tools mentioned in the upcoming Linux section for instance are available for all the BSD's.

5.3. Shell

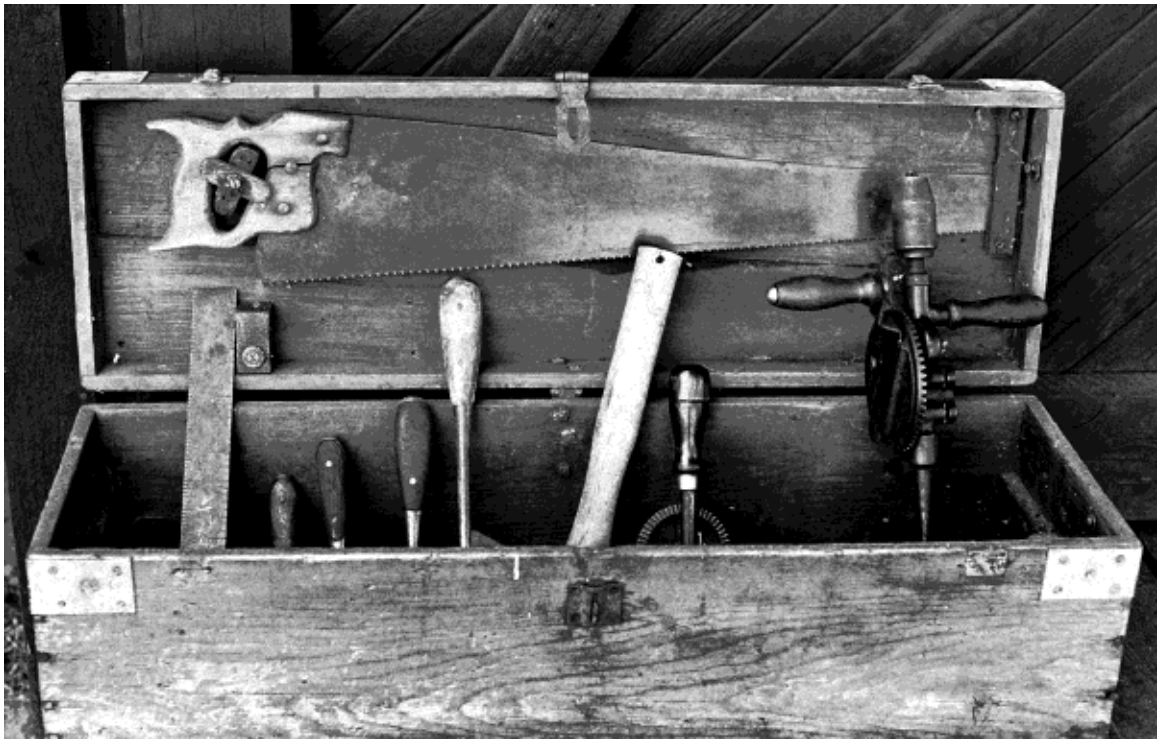


	OpenBSD		NetBSD		FreeBSD	
name	src	man	name	src	man	name
sh			15,531	39	13,330	31
ksh	19,535	54	22,661	48		
csh	13,310	27	14,253	27		
tcsh					57,327	83

OpenBSD and NetBSD use variants of the public domain `ksh` and `csh`, also known as `pdksh` and `pdcsh`. NetBSD's `sh`, the Almquist shell, also known as `ash`, is the basis for `dash` in Linux. The C shell was developed by the BSD community before the Bourne shell was introduced in UNIX V7 in 1979. It championed command history, jobs and other interactive features that we now take for granted. Unlike the Bourne shell however, its main development focus wasn't on providing a programmable shell, and suffers in comparison as a basis for shell scripting. The new and improved `csh`, called `tcsh`, is used by FreeBSD and DragonFly, and it's arguably on these platforms alone that the C shell is still in active use.

In practical usage `ksh` is very similar to `bash`, but shell scripts may not be fully compatible, in particular array handling is different. There is a good Perl script for checking shell scripts for `bash` idiosyncrasies, and suggesting more portable syntax, called `checkbashisms`. A quick Google search should located it quickly enough. The Public Domain `ksh` is about 20 times smaller and its manual half the size of `bash`. Ksh93 on the other hand is nearly 10 timer bigger then `pdksh`.

5.4. Applications



OpenBSD			NetBSD			FreeBSD		
name	src	man	name	src	man	name	src	man
echo	27	1	echo	43	0.5	echo	122	1
cat	199	1.5	cat	268	1.5	cat	332	2
ls	1006	5	ls	1269	5	ls	2620	9
find	1862	7	find	2164	7	find	2896	11
cp	570	2.5	cp	681	2.5	cp	797	3
wc	215	1.5	wc	259	1.5	wc	276	2
sed	2409	6.5	sed	2567	6.5	sed	3082	7
awk	5489	7	awk	7835	7.5	awk	5480	7
tar	7912	4.5	tar	9343	4.5	tar	7748	15
sort	4220	6	sort	1551	3.5	sort	4955	7
tail	838	1.5	tail	831	1.5	tail	1224	2
ps	1427	8.5	ps	2441	7.5	ps	2290	9
file	3280	1.5	file	47,959	8.5	file	48,537	9
grep	1361	3.5	grep	1627	7.5	grep	1708	8
date	204	2.5	date	437	2.5	date	1014	5
bc	1409	4.5	bc	5221	15	bc	1419	5
less	11,769	25	less	24,560	30	less	19,074	31

In total the `/bin` and `/usr/bin` directories in OpenBSD contain about 390 utilities, the source code for these programs count 600,000 lines of code, and section 1 of the manual covers 5000 pages. For NetBSD there are about 540 utilities in these directories, but only half are actually maintained by the NetBSD developers, the rest are imported from external sources. Of the ones that NetBSD maintains, its source code numbers 310,000 lines of code. Section 1 in the manual covers 4600 pages. (most of these core utilities have been imported to Minix) FreeBSD has about 530 utilities, it maintains about two thirds

of these themselves. Those have a combined source code of 250,000 lines. Section 1 in the manual covers 2900 pages.

At this point we can begin to see some differences diverging between the BSD's. Many of the OpenBSD programs listed above are significantly simpler then their counterparts in NetBSD and FreeBSD. `echo` and `date` are nearly five times bigger in FreeBSD, `ls` nearly three times as big. Of course most of these utilities are still very small compared to Linux. FreeBSD `grep` is 100 times smaller then the GNU version, `find`, `sed`, `awk`, `tar`, `ps` and `bc` are other examples where the GNU versions are huge by comparison. And yet the BSD utilities do basically the same job, in fact in my experience they are often more robust then their Linux counterparts. In the case of OpenBSD especially a lot of utilities are included which either don't have an equivalent in Linux, or it replaces very poor alternatives. Some of these applications, such as `ssh`, `tmux` and `nc` have seen mass adoption. Programs in the table below that is only available in OpenBSD are marked with an asterix.

More Applications

	OpenBSD		
name	src	man	comment
bsdgames	75,053	-	Old games (actually maintained in OpenBSD)
ssh	74,390	18	Remote shell, used by *everyone*
sftp	5,749	3.5	Secure FTP, part of the ssh suite
scp	1,364	7	Secure rcp, part of the ssh suite
got*	114,377	24	Game of Trees, alternative to git (tog alt. to tig)
tmux	43,149	54	Terminal multiplexer, alternative to GNU screen
opensmtpd*	37,914	1.5	Email server, alternative to sendmail etc.
httpd*	10,486	1	Web server, alternative to apache etc.
systat	7840	8	System statistics, alternative to htop
openrsync*	5547	3	Sync files over the net, alternative to rsync
aucat*	3355	3	Audio player/recorder/mixer, alternative to sox
cdio*	2727	3	CD player/ripper/burner, alternative to cdparanoia/cdrecorder
nc	1845	6.5	NetCat, a swiss-army tool for the web
doas*	884	1	Run commands as another user, alternative to sudo
units	589	2.5	Unit conversion
lam	221	1	''Laminate'', ei. concatenate files side by side
fold	184	1	Fold lines, alternative to fmt
vis	175	1.5	Show invisible characters in input
rev	76	0.5	Reverse characters in input

Aggressive surgery is the only known treatment against the cancer of feature creep. Yet few developers outside of OpenBSD have the guts and discipline required to rescue their code. Of course with healthy living you can often avoid the problem preemptively, and adhering to the UNIX principal of ''worse is better'' will keep feature creep to a minimum. To illustrate, the `file` utility is used to figure out what kind of filetype a given file is. This is really a compromise, since the UNIX operating system itself doesn't know or care about filetypes. The utility scans the first couple of bytes of the file and makes an educated guess. The process is slow and error prone, but here is an even bigger problem: There is no end to obscure filetypes out there, and new ones are invented every year. The only sustainable and wise course of action is to follow the principle ''worse is better'': Don't even try to guess every filetype known to man 100% correctly, just look for usual suspects and do a reasonably good job. OpenBSD follows this principal, FreeBSD (and NetBSD) does not, none of these versions are fast, and none are able to guess filetypes 100% correctly, but the OpenBSD command is at least 15 times smaller and that much faster and more secure. These numbers give some indication of why OpenBSD users are so passionate about their OS, it isn't just the worlds most secure operating system, but arguably the simplest and most elegant modern UNIX alternative too. No other UNIX operating system have developers with such a hardline approach to quality and simplicity.

Nevertheless idealism comes at a cost, OpenBSD is not built for heavy loads and supersonic speeds, and it is much easier to port applications to FreeBSD (or NetBSD) then OpenBSD. While there are a decent 10,000 ports available for OpenBSD, FreeBSD has nearly 30,000 (NetBSD has nearly 20,000)! And remember we are talking about source code projects here, the number of binary packages will be twice this size. Many popular applications and features such as Wine, Skype, Steam, Flash, VirtualBox, multilib support, and Linux and Solaris emulation (necessary to support ZFS) are only available on FreeBSD (and NetBSD). And FreeBSD is the only BSD alternative that comes with desktop oriented distributions, such as TrueOS and GhostBSD. If Linux is the ‘‘Windows-killer’’, then FreeBSD is the ‘‘Linux-killer’’. It can replicate your Linux workflow with ease while providing a server that even rivals Solaris! (of course OpenBSD users avoid FreeBSD for precisely the above mentioned reasons)

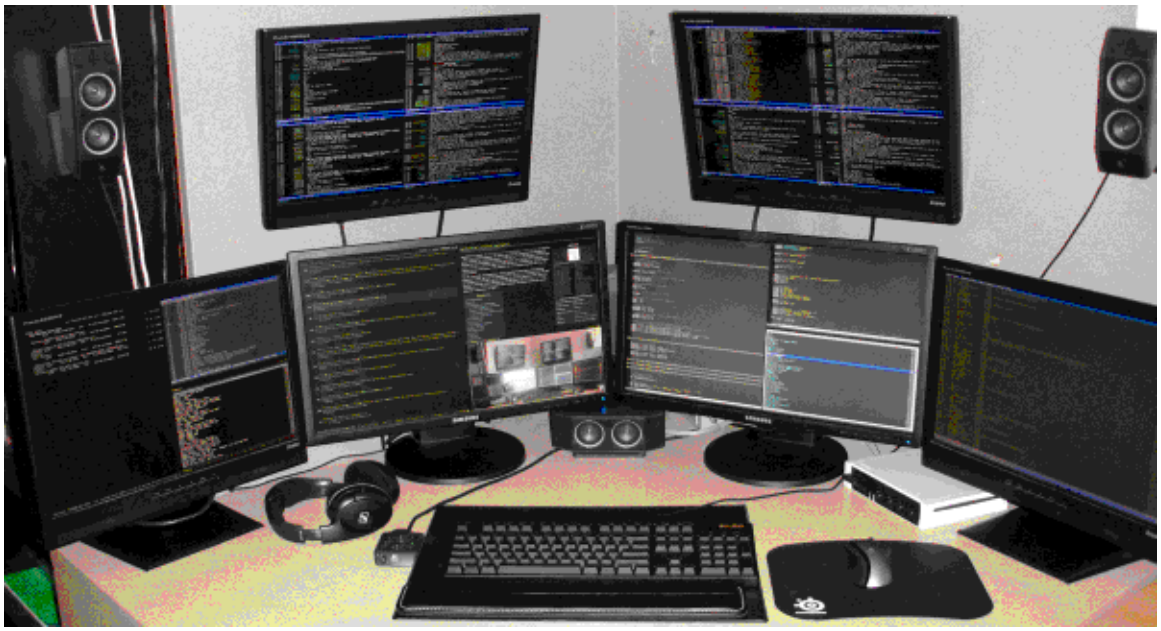
The parentheses around NetBSD here are intentional. All the limelight usually goes either to the *friendly fatty*, or the *obnoxious onslaught*, so it’s easy to forget that there is a third* alternative: the *nifty nerd*. The NetBSD folk have an extremely pragmatic and down-to-earth approach to development. If some external tool is good enough, use it. If the problem isn’t too big, don’t worry about it. When these developers do take action, they do so with quiet professionalism, doing a phenomenal job without anyone noticing. As Michael W. Lucas once said: *If the NetBSD guys start a secret ninja club, we will all be dead without knowing it.* NetBSD may only have two thirds as many ports in their repo as FreeBSD, it may only have half the security mitigations of OpenBSD, but it still is a darn good operating system! It is significantly simpler then FreeBSD, but provides much more applications, and is easier to port software to then OpenBSD. It also has some unique tricks up its sleeve, from kernel Lua scripting, cross-compiling auto builds and reproduceable binaries, to a Ports Collection that works across multiple operating systems (pkgsrc is in fact a superb addition to Haiku and Solaris!). Lastly it’s quite famous for its absurd level of support for obscure hardware. If you want to put BSD on your toaster, there is only one candidate!

Superficially though all the BSD variants are very similar, and it’s not uncommon for a BSD user to use several variants in his workflow. In fact the only differences from a BSD desktop and a Linux one, is entirely under the surface. The BSD’s have simple core utilities of high quality with readable manpages, and the system is very well organized. For example sources for `/bin/cp` and `/usr/bin/wc` is `/usr/src/bin/cp` and `/usr/src/usr.bin/wc`. respectively. All 3rd party applications are installed in `/usr/local`, providing a clean separation between base and ports. The Ports Collection itself is well organized and it’s easy to inspect and modify. System configuration is much cleaner, with nearly everything centralized into a single file: `/etc/rc.conf`. On the surface Linux can give you the same user applications, but if you peek under the carpet things are decidedly more messy! Since software usually comes first to Linux, the BSD repos may be missing some of the latest updates or proprietary offerings, but this is a small price to pay for the added stability, cohesiveness, documentation and overall quality that these systems provide.

FreeBSD comes with a very good Handbook which should get you up to speed quickly, NetBSD also has a good User Guide. OpenBSD has an informative FAQ, and you might want to invest in *Absolute OpenBSD* from M. Lucas, if you’re planning on using the system seriously. Another pro book tip for all the BSD’s is the old classic *UNIX Power Tools* from O’Reilly.

* And a fourth, DragonFly BSD (*daring deadbeat*?), which even the NetBSD users tend to forget about...

5.5. Desktop



OpenBSD			NetBSD			FreeBSD		
name	src	man	name	src	man	name	src	man
twm	29,012	24	twm	29,001	24	Lumina	442,308	14
fvwm	46,749	51	ctwm	32,035	51			
cwm	5637	3						

FreeBSD and DragonFly does not come with a desktop by default. Lumina was created by the TrueOS developers, in response to the ever increasing effort required to porting KDE to FreeBSD. It is fairly similar to LXQt, and it's the only desktop beyond window managers that has come out of the BSD camp. Having that said all of the BSD's include dozens of desktops and window managers from Linuxland in their Ports Collection. Only the biggest and most unreasonably obtuse desktops aren't always available, it took years for the FreeBSD Ports Team to import GNOME 3 and KDE 5 for instance, and it was absolute hell for those poor devils! KDE 5 isn't even ported to OpenBSD yet (due to Wayland dependence).

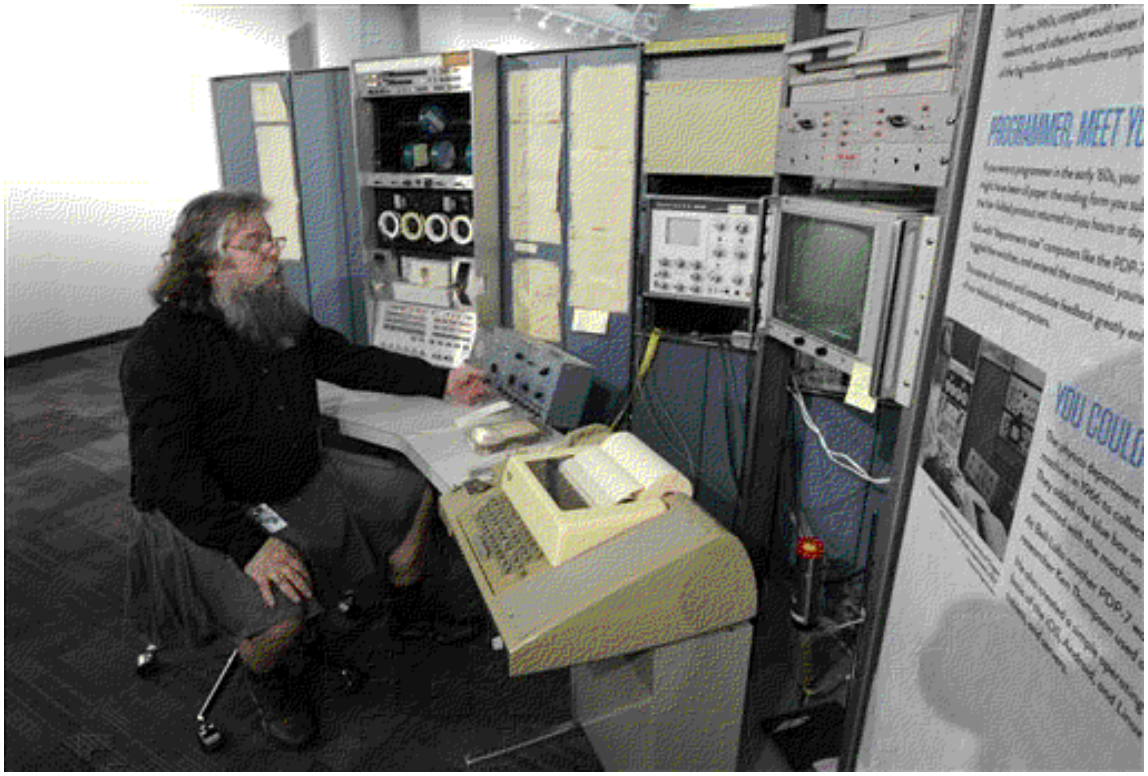
The default OpenBSD desktop, `fvwm`, is the old version, not `fvwm2` that is all the rave nowadays. `cwm` is inspired by `evilwm`, but rewritten from scratch, presumably because it wasn't quite evil enough.

I must admit, I was quite biased against the NetBSD desktop, presumably because previous experience with `ctwm`, which NetBSD uses as its default desktop, had left a bad taste in my mouth. The memory usage of NetBSD was also atrocious, using more RAM then Debian running Xfce! The reason for this high memory usage though, is that NetBSD uses a significant slice of available RAM as temporary storage for `/tmp`. Once that is taken into account, the memory usage is not much worse then its competitors. And the developers have even managed to pull off a fairly tasteful `ctwm` setup. Most users will probably find it uninspiring though, but as mentioned, there are plenty of 3rd party options available.

Naturally all these window managers depend upon the X Window System, OpenBSD ships with its own fork, called Xenocara, but it's virtually identical to Xorg. The combined source code of Xenocara is nearly 10.2 million lines of code, twice the size of the kernel. Manpages for Xenocara span 5900 pages,

more then all manpages for general user commands combined. What gain is there in a simple and light-weight window manager, when it has to run on top of the Cthulhu monstrosity that is X?

5.6. Programming



Traditionally BSD systems used the GNU Compiler Collection like everyone else, but in recent years FreeBSD and OpenBSD have switched to the new rival Clang/LLVM compiler suit, whereas NetBSD and DragonFly stick with the old GCC. Both are about 10 million line monstrosities, but doing something useful with them require that you learn the system libraries in addition, covered in section 3 of the manual, in OpenBSD they span 2100 pages, in DragonFly 10,800 in NetBSD 14,500 and in FreeBSD 26,300 pages. The beautifully craftet code of the small BSD utilities, stand in stark contrast to the overall complexity of a modern C development environment.

Perl is also included in OpenBSD and used various places throughout the system. The package manager is written in it for instance. NetBSD includes Lua by default. Of course you can find pretty much any programming language or tool you want in the repository, so virtually any kind of development workflow in Linux can be reproduced in all the BSD's. A good book tip for traditional C programming in BSD, besides *The C Programming Language*, is the classic *Advanced Programming in the UNIX Environment.* Programming in C is especially nice in BSD, because you have easy access to a wealth of high quality source code. Just reading the source code of basic utilities like `ns`, `cat`, `grep`, etc, is a very good way to become a proficient systems programmer.

5.7. Kernel

The total manual pages for section 2 and 9, covering system calls and kernel internals, is about 640 pages for OpenBSD, 5080 for DragonFly, 8430 for NetBSD and 16,000 for FreeBSD.

	OpenBSD	**NetBSD**	**Minix**	**DragonFly**	**FreeBSD**	
name	src	src	src	src	src	comment
boot	3645	3758	-	10,218	2594	startup procedures
sys	19,563	33,536	11,385	32,654	47,852	headers
kern	71,789	132,855	51,952	108,156	201,693	kernel facilities
ddb	64,195	5003	49,802	3288	22,792	debugging, testing
compat	-	158,620	-	-	147,951	compatibility layers
crypto	9513	18,207	-	12,464	273,113	cryptographic support
security	-	2441	-	-	29,530	various security features
dev	4,676,156	2,566,912	113,318	2,790,450	3,343,354	device drivers
fs	66,122	219,960	9793	139,031	702,554	filesystems
net	141,005	210,900	15,078	253,965	482,410	network
misc	30,504	847,948	122,707	175,465	862,907	miscellany
arch	443,029	1,545,613	-	36,298	560,015	architecture dependent code
sum	849,365	3,178,841	260,717	771,539	3,333,411	sum without drivers
all	5,525,521	5,745,753	374,035	3,561,989	6,676,765	sum with drivers

Beasty is in the details, as they say, and these statistics reveal some major differences between the various BSD's. OpenBSD has even greater driver support then FreeBSD in addition to supporting many obscure architectures, but the kernel itself is minuscule, if you exclude device drivers and architecture dependent code it is not much larger then Minix! This simplicity lies at the heart of OpenBSD's fame to security. For mitigations such as `pledge`(2) and `unveil`(2) to be effective, the entire source tree needs to be rewritten. This is done with relative ease in OpenBSD, but it would require unbelievable amount of effort to consistently incorporate into NetBSD or FreeBSD, not to mention Linux. It also explains OpenBSD's resistance to ZFS, importing this filesystem would effectively double the size of the kernels core. The developers are aware that something needs to be done with the archaic filesystem support, but as of yet it is unclear in which direction OpenBSD will go.

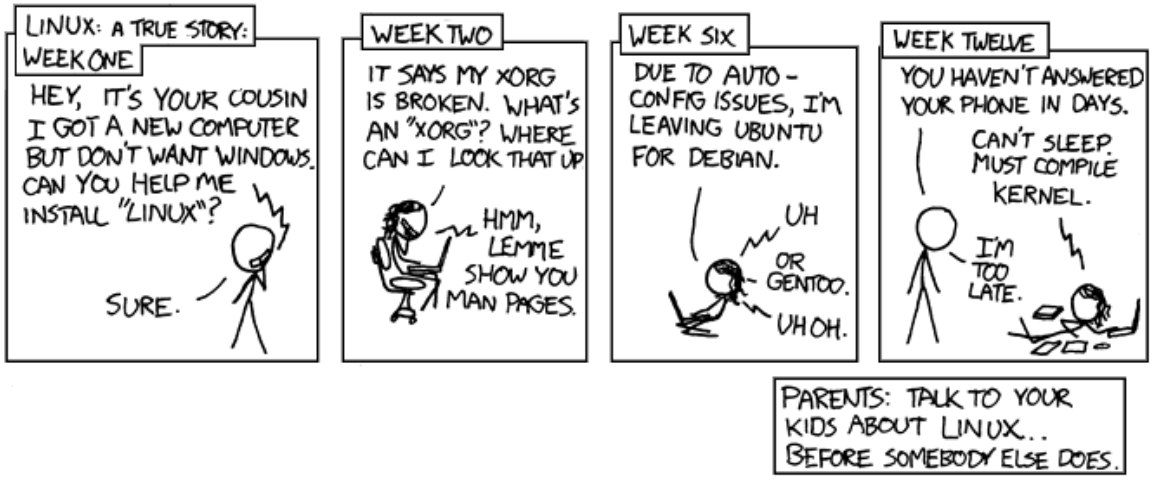
As for NetBSD, it sits somewhere between OpenBSD and FreeBSD when it comes to kernel features. It has unprecedented support for obscure architectures, and very good compatibility support for various binaries. And yet support for popular hardware is actually quite bad, and the extensive compatibility layers are a security nightmare. *Of course it runs NetBSD,* as the popular slogan goes, but perhaps we might add: *unless you're using normal hardware.*

Although FreeBSD initially focused on the PC platform alone, it has since been ported to many different architectures. The import of ZFS from OpenSolaris in recent years, accounts for the huge increase in filesystem source code. The FreeBSD developers are in many ways trying to compete with Linux, and thus the majority of their attention is devoted to expanding kernel features, to support vendor applications and compete in the enterprise market. To a much greater extent then the other BSD's, the practical userland is delegated to the Ports Team (which in all fairness is doing a good job), one example of this is that there is no graphics support at all in FreeBSD base.

As for DragonFly, it split from FreeBSD way back in simpler times when the dinosaurs thought wireless roaming was a nifty idea, and they are getting a bit long in the tooth. One big innovation though is their Hammer filesystem, it has some of the features of ZFS, but is far simpler. Yet despite being a quasi-research project, DragonFly BSD can operate as a general purpose operating system. Minix however is purely a research operating system, and unless you are developing a product on embedded devices, it would be hard to use it on real hardware doing real work. The kernel itself is actually just 18,000 lines of code, and it runs the filesystem, networking and so on as userspace applications. This gives the operating system tremendous stability, if the filesystem crashes for instance, the kernel will just restart the filesystem application and carry on.

Minix is very well documented in Tanenbaum's book *Modern Operating Systems, 3rd Edition* although the text is a little bit out of date. As for FreeBSD a good textbook discussing its internals is *The Design and Implementation of The FreeBSD Operating System* available from Addison Wesley.

6. LINUX and SOLARIS



In the early 80's UNIX had existed for over a decade and a surprising amount of variants existed all over the place, AT&T made an effort to centralize all these forks and their innovations, including many from the BSD camp, into one unified system. This work eventually resulted in a commercial UNIX called System V, with this base many corporations built their own proprietary varieties of UNIX with AT&T's blessing. Solaris from Sun Microsystems was one such commercial flavor, arguably the most successful of them. With the rising popularity of Linux, Sun decided to opensource their operating system in order to better compete in this growing market, and released OpenSolaris in 2005. Although Linux tries to mimic System V closely, making these two systems very much alike, the relationship between their respective communities remained strained. Whether this was due to decades of corporate evilness from Sun, or the religious fanaticism of the GNU freedom fighters, is hard to say.

In any event before OpenSolaris had really taken off, Oracle bought Sun. What then ensued might perhaps best be described as *the Silicon Valley chainsaw massacre.* Oracle basically went in and tore everything a-Sun-der, doing its best to sabotage all of their projects. One can only speculate as to the business strategy behind this mayhem, but the result was that a number of community forks, such as LibreOffice and MariaDB, sprang up and saved some of the former Sun projects from the flames. Illumos was one such community effort that picked up the pieces from OpenSolaris and continued its development. Today there are several distributions of Illumos, most of which are very alike* but are aiming for different goals. OpenIndiana is a very feature-full distribution providing both desktop and server solutions, whereas OmniOS focuses exclusively on the server market. In later years there has been much collaboration between Illumos and FreeBSD, and Solaris technologies such as ZFS and DTrace have been ported to FreeBSD and NetBSD, whereas Linux adoption of these tools has been slow.

*) the term Illumos *distribution* may confuse Linux users. Illumos isn't just a kernel, but a complete operating system including userland and libraries. So Illumos distributions all share a fairly large common base.

Larry Ellison, CEO of Oracle

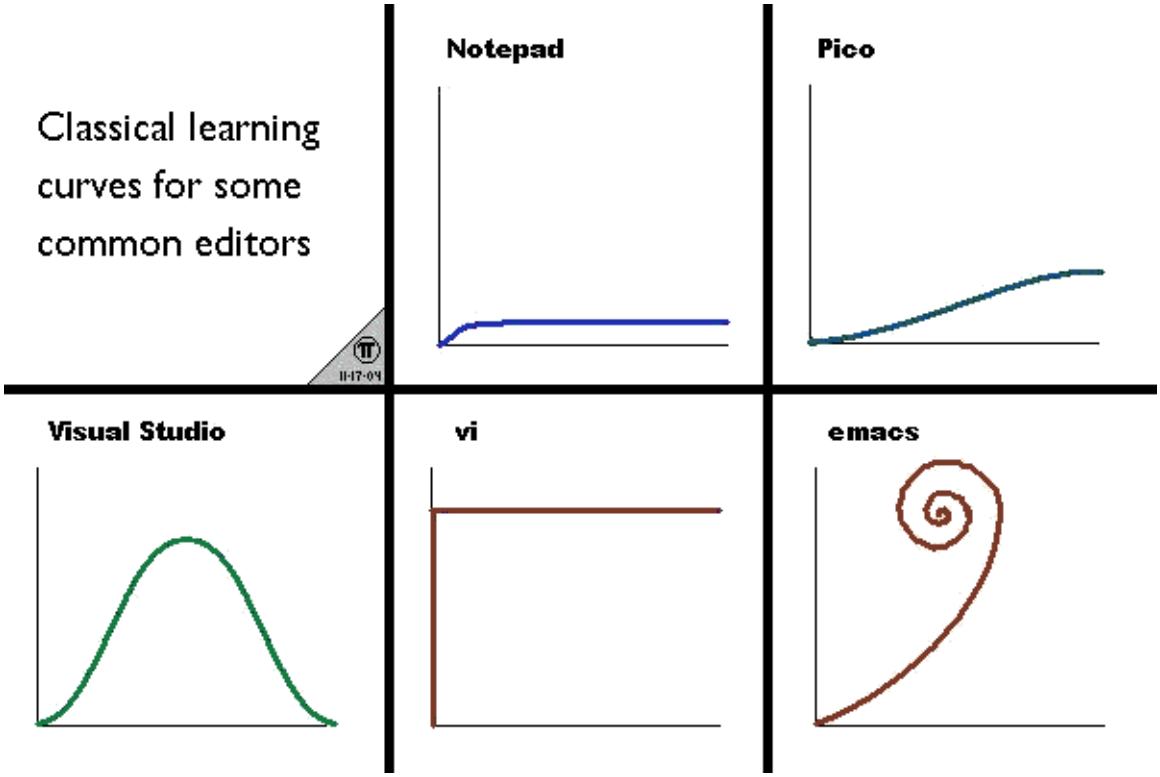
Linux is by far the most popular opensource operating system today, unlike many of the other candidates here it does not originate directly from UNIX, but was written from scratch. Nevertheless it is absolutely a UNIX clone, as it tries to mimic this operating system in minute detail. Also in contrast to most operating systems, Linux is only a kernel, the core component responsible for handling hardware devices, allocate memory and CPU resources between programs, providing a filesystem, networking and so on, but doesn't provide any user applications.

To get a practical system, the Linux kernel must be distributed with a userland and libraries. Usually, these are made up of the GNU ''operating system'', which is unique itself in that it has everything except a kernel. Both these projects complement each other nicely and makes the practical ''Linux'' (or GNU/Linux as the GNU people like to call it) operating system possible. Of course modern distributions are usually bundled with quite a few additional extras, such as a fancy desktop and web browser and what not. The job of combining all these disparate projects into one cohesive system is delegated to independent *distros.* There are several hundred such distributors of Linux, some are backed by multi-million dollar companies, and some are hacked together by geeks with too much spare time on their hands. The ones included here are the classic distros Debian and Slackware, free counterparts to the commercial distros Red Hat and SUSE, ie. AlmaLinux and openSUSE. The complete decoupling of the Linux kernel with its userland and libraries gives the operating system unusual flexibility. This is exploited by many proprietary systems, such as Android. We will not analyze that system here, but we will look at the minuscule Tiny Core and Alpine Linux that does away with all the GNU stuff and opts for a much smaller environment.

The majority of userland applications in Illumos, and indeed BSD, are imported from Linuxland, so unless there are unique Illumos programs of interest only Linux applications will be listed. And since there are no ''standard'' applications in Linux, we will just list a handful of popular choices. The ones that have been ported to OpenIndiana* will be marked with an asterix (all of them have been ported to the BSD's).

*) Technically Illumos is a fork of OpenSolaris, which is a fork of Solaris. But these systems are so alike that I will often use their names interchangeably. ''Solaris'' is less specific then ''Illumos'' though, ''OpenIndiana'' is very specific.

6.1. Text Editors



name	src	man	name	src
ed*	3059	1.5	editor	526
ne	47,704	1.5	mousepad	112,168
jed	64,669	5	gedit*	352,996
joe*	95,044	62.5	geany*	411,372
elvis	116,253	9	kate/kwrite	495,744
nano*	196,407	7.5	kdevelop	914,587
vim*	1,001,639	8	qt-creator	2,219,929
emacs*	1,860,600	8.5	netbeans*	7,441,863

Mousepad from the Xfce desktop is probably the closest equivalent to Notepad for Linux. Gedit and Kate (or the more simplistic Kwrite) are the standard editors for Gnome and KDE (Mate uses a fork of Gedit called Pluma). There are also a handful of more elaborate IDE's for Linux, although none can really compete with Microsofts Visual Studio. As a basis for comparison, the popular Notepad++ editor on Windows is 514,578 lines of code, which is about on par with the simple GUI programming editors for Linux, but only a fraction of the size of vim, or Emacs. Windows developers frequently lament the lack of a good alternative to Visual Studios in Linux, but there are many candidates that are at least halfway there, `vim` being one of them. The old joke about Emacs being an operating system that only lacks a good text editor, does have some basis in reality as far as complexity is concerned. UNIX was created because Multics was too bloated with its whooping 300,000 lines of code. GNU Emacs is 6 times bigger then that.

Despite the large collection of alternative editors, it is surprisingly hard to find a simple text editor for Linux. The Tiny Core Notepad-like FLTK ''Editor'', is a pleasant exception. As is the Busybox implementation of `ed` and `vi` included in the distro, having 751 and 3426 lines of code, respectively.

But one may wonder; is it really possible to develop good code with such simple tools? Yes. Writing good code is hard, the only way to do so is to think until understanding dawns. When code breaks, step away from the keyboard! Think. Explain the code to a stuffed teddy bear in close proximity (no really!). Go outside, take a walk. Under the best of circumstances, using a sophisticated IDE does not significantly reduce the effort needed to understand your code, and no amount of sophistication can help the debugger develop the needed characteristics in his programmer. To quote Pike & Kernighan:

As a personal choice, we tend not to use debuggers beyond getting a stack trace or the value of a variable or two. One reason is that it is easy to get lost in details of complicated data structures and control flow; we find stepping through a program less productive than thinking harder and adding output statements and self-checking code at critical places. Clicking over statements takes longer than scanning the output of judiciously-placed displays. It takes less time to decide where to put print statements than to single-step to the critical section of code, even assuming we know where that is. More important, debugging statements stay with the program; debugging sessions are transient.

For serious programmers, I highly recommend the old book from which this quotation is taken, *The Practice of Programming.* The tools we use today may have become more complicated, but the essential skills needed by a programmer are still the same. If anything, these skills have become increasingly important as the world around us has grown more convoluted.

6.2. Office

name	src	man	name	src
groff*	209,547	14	xpdf	122,623
texlive*	7,831,989	335	evince*	258,612
diction	10,535	2	okular	435,321
WordNet	24,773	5.5	abiword	888,930
ispell	14,383	16	gunmeric*	1,861,343
aspell*	88,445	34.5	calligra	4,057,156
hunspell*	102,616	11.5	libreoffice*	9,243,483

The numbers for KDE's Calligra office suit does not include the sources for Krita, arguably its main claim to fame. The suit contains a great many programs, but most are somewhat crude in comparison to more mainstream office suits. In contrast, the Gnome office suit consists of only two applications, the word processor Abiword, and the spreadsheet Gnumeric, but these work very well. LibreOffice, the OpenOffice fork, is of course the big one, and the only candidate that is really a viable alternative to Microsoft Office. It does a relatively good job in this respect, it is quicker, has better file format, scripting and platform support, let alone good documentation and an open development model. But it obviously isn't 100% compatible with Microsoft Office, which seemingly is sufficient reason to condemn the whole project as obscure by the general public. Happily Gnome office provides an even less compatible suite, which is that much snappier and more robust.

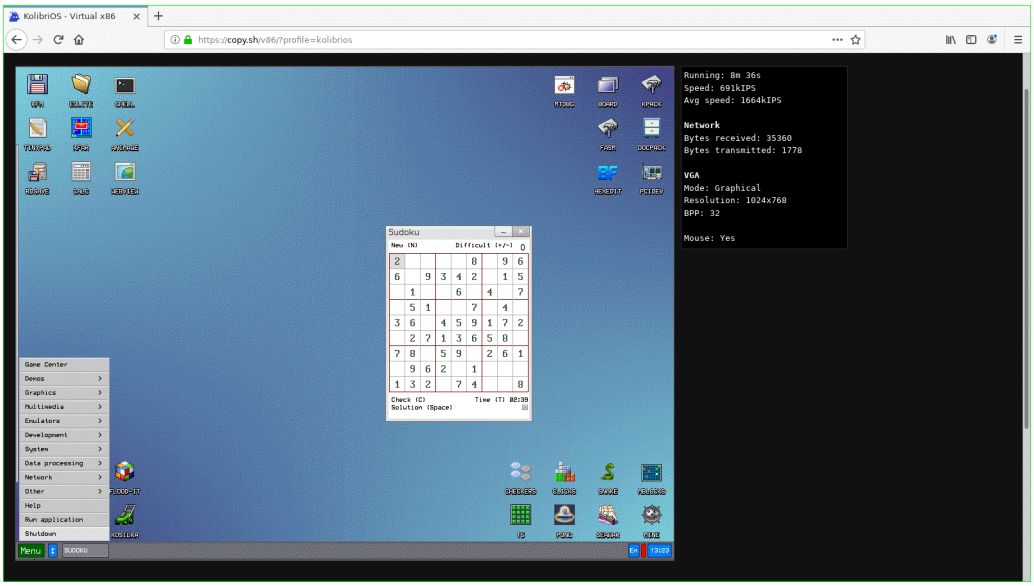
There are many more office tools in Linuxland we could have mentioned, and for a casual user there should be plenty to choose from whatever your needs are. GNUCash for managing finances, Scribus for designing magazines, Lyx for writing LaTex documents and Inkscape for vector graphics, are some popular productivity applications. But for serious book publishing, there is sadly no opensource offering that can compete with large proprietary suits, such as those from Microsoft or Adobe. It is sad enough that the world has grown dependent on overblown software, double so that its addicted to proprietary solutions.

Speaking of bloated software, GNU troff, or `groff`, is huge if you compare it to the old UNIX `troff`. Its obesity is that much more serious as it doesn't handle itself very gracefully, especially when dealing with unicode input. But these problems pale in comparison to modern Tex (which also has unicode issues). The author of Tex was mightily impressed with `troff` way back in the 80's, it only lacked one or two features that he wanted. 40 year later all the bees in the world died because of the massive burden Tex had on the environment. This tragic story is entirely typical of a software projects normal life cycle.

6.3. Internet

name	src	man	name	src	name	src
links*	96,383	5	netsurf	299,306	lighttpd*	99,312
w3m	101,171	10.5	falcon	323,500	nginx*	175,208
lynx*	283,061	22.5	qt-webkit	1,702,124	apache*	555,430
mailx*	92,578	157	firefox*	31,236,630	exim	133,615
mutt*	260,907	4.5	chromium	33,674,513	postfix*	219,151
irssi*	102,523	1.5	clawsmail*	644,993	dovecot*	544,130
weechat	584,075	4	thunderbird*	32,994,427		
rsync*	74,581	89	hexchat	250,282		
wget*	309,642	38	pidgin*	1,232,782		
curl*	329,184	96	transmission*	530,372		

Modern web browsers have become almost inexplicably complex, it's on par with the biggest operating system kernels and desktops, and in truth it has absorbed the roles of both.* If you have both Firefox and Chrome running on your Linux GNOME desktop, you are essentially running three desktops and three operating systems simultaneously, Linux and GNOME being the lightweighters. To put things into perspective, lets imagine that the Firefox source code doubles in size the next decade. The browser would then be big enough to incorporate the functionalities of the multimedia player VLC, the 3D editing suite Blender, the KDE office suite Calligra, the image editing suite Gimp, the Xfce desktop with all its applications, the FreeBSD kernel with all its drivers, the Battle of Wesnoth game, in addition to SuperTuxKart, DosBox and all of the KDE games, with room to spare! In actuality Firefox has grown by 405% in the last 10 years, if it continues to grow at the same speed, by the early 2030's it will be twice the size of KDE Plasma desktop with all of its applications, and five times the size of the OpenBSD operating system with all of its components. One can only imagine the horror 10 years after that.



A good OS in a bad OS in a good OS

Isn't that a good thing? After all everyone has a browser, right? No. Browsers are not automatically portable because they are popular. On the contrary, as web browsers grow increasingly complicated, the effort to support them on various operating systems increases. The more complex a browser becomes, the *less* ubiquitous it will eventually be. If fact we are already seeing this. Alternative operating systems, such

*) Not only are there fully fledged browser based operating systems, such as FriendOS, but there are fully fledged browser based virtual machines from which you can run ''normal'' operating systems (which can also run web browsers themselves (which can probably also run operating systems (and so on ad infinitum...)))

as the BSD's, are struggling to keep up. Even Opera and Microsoft eventually abandoned their browsers, and just copy pasted Chrome, because it was just *too much work.* It is telling that Microsoft do not have the resources to make a web browser these days! Small wonder then that the once mighty Firefox is now loosing the fight. A cynical man might say that it's poetic justice that the Mozilla monster is being slowly crushed by the complexity it helped to create, but I just think it's sad.

Besides, if we accept that the web browser should take over the responsibility of our operating system, it is only fair to compare the quality and overall design elegance of these browsers to real operating systems. But since I have managed to refrain from cursing so far in this article, I think it's safest not to make any such attempt. Lets just say that if anyone believes that there is grace and wisdom in the code of their favorite browser, I cordially invite them to read the source code and find out. I'll get back to you when you are done, say, in 30 years or so.

6.4. Shell

Linux			**Illumos**		
name	src	man	name	src	man
es	8576	30			
dash	20,923	29	sh	8959	-
tcsh*	69,487	82	csh	13,108	37 pages
zsh*	188,621	7	-		
ksh93*	198,140	64.5	ksh	73,509	67.5 pages
fish*	362,148	-			
bash*	381,782	116.5			

Both Solaris and most Linux distros use `bash` as their default shell, although Solaris also ships with `ksh93`. Some Linux distros use `dash` as their default shell. Though not necessarily faster then `bash` this shell has strict POSIX compliance, and is the only popular alternative which in any way can be referred to as ''light''. `zsh` and `fish` are popular choices among hipsters and newbies, respectively. While `tcsh` and `es` are only used by weirdos, such as FreeBSD and Plan 9 fans. There are even more obscure shells out there, for people weirder then me, if you can imagine!

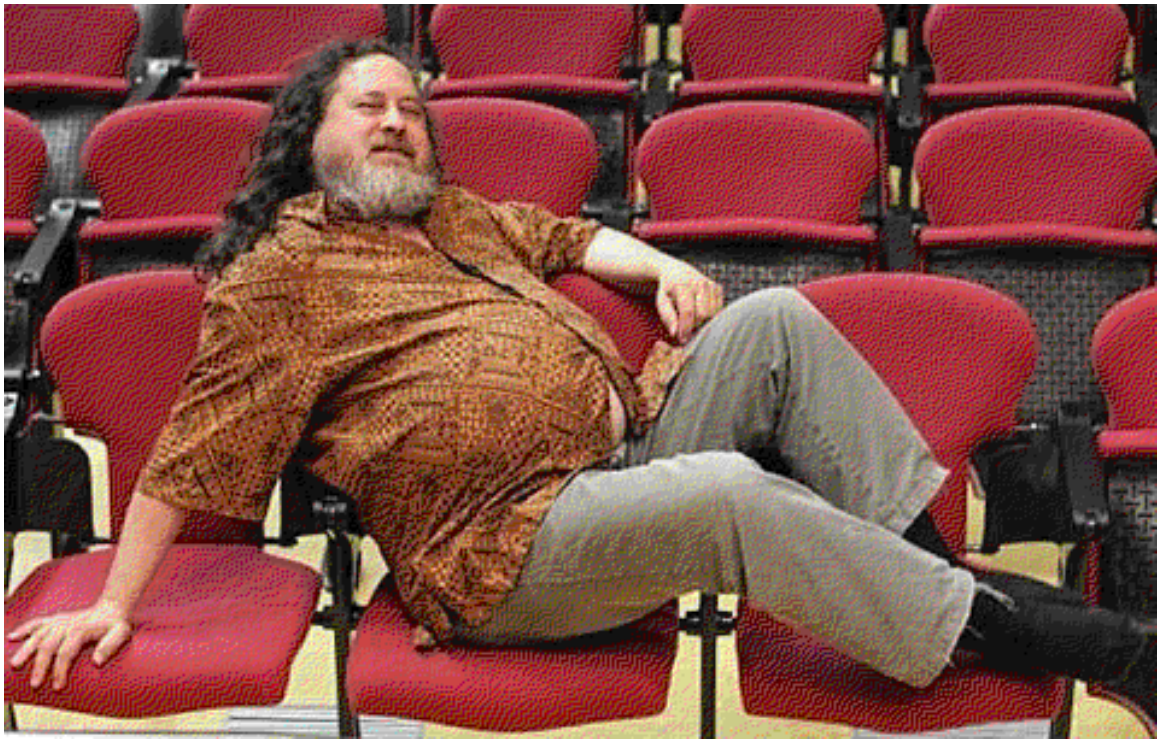
In popular Linux distributions a user can install and run their operating system without ever having to dip their toes into the murky water of the command line, and no doubt this is one of the reasons why Linux is so much more popular then, say OpenBSD. Nevertheless it is the shell more then anything that separates UNIX from other operating systems, and it is only by using it that the system stops being a mere toy and starts to become a serious tool. Especially in Linux the wealth of command line tools are overwhelming, to the point of being ridiculous. It is only on the Linux console that you can watch videos, or have spinning ASCII 3D desktop cubes, for instance.

And yet strong conservatism hinders important progress. Why do modern shells emulate teletypes from the 70's? Why isn't GUI's more integrated with the command line? Why is unicode *still* not universally supported? Unlike most operating systems, the kernel, the system libraries and the userland applications, such as the shell, in Linux are completely decoupled, so there is no valid reason to refrain from experimental innovation. Sadly though the iron wall of backward compatibility prevents that. In my humble opinion, none of the shells listed here can compare to the elegance and power of the Plan 9, Inferno or SerenityOS shells. Small wonder then that fewer and fewer people are interested in the command line.

Yet there is hope. Alpine Linux throws out the GNU userland and its glibc in favor of the minuscule BusyBox and musl library. Tiny Core follows a similar course, but focuses more on creating a RAM-only system with a minimal GUI, using a custom fork of X and the tiny FLTK toolkit. These fairly recent projects, and others like them, display a welcomed out-of-the-box thinking. Yet, their simplicity is mismatched by the comparatively bloated Linux kernel. In fact, the kernel source is about two orders of magnitude bigger then the entire Alpine/Tiny Core userland. What would really have been interesting to see is a tiny fork of the Linux kernel, 10 or 20 times smaller then the default, coupled with BusyBox and musl. Such a tiny distro wouldn't just be on par with the BSD's, but on par with Plan 9! Such simplicity would likely lead to much greater flexibility and innovation. Alpine and Tiny Core are absolutely a step in the right direction, but further steps need to be taken before Linux can really live up to its potential.

6.5. Applications

GNU			**Busybox**	**Illumos**	
name	src	man	src	src	man
echo	212	1.5	163	94	4.5
cat	516	1.5	102	391	5.5
ls	4055	4.5	825	2441	31.5
find	10,636	30.5	1029	1768	12.5
cp	1019	3	135	-	6.5
wc	773	1	139	-	3.5
sed	168,932	5.5	1042	1912	8
awk	203,099	39.5	2631	5962	26
tar	322,179	21	812	6793	16
sort	3365	3	412	4882	9.5
tail	1774	2	278	1760	3.5
ps	4924	21.5	523	2994	15
file	51,725	9		2829	4
grep	197,735	12	642	1107	5
date	503	4	204	325	5.5
bc	27,053	14.5	5925	882	5
less	27,537	33.5	1466	-	34.5



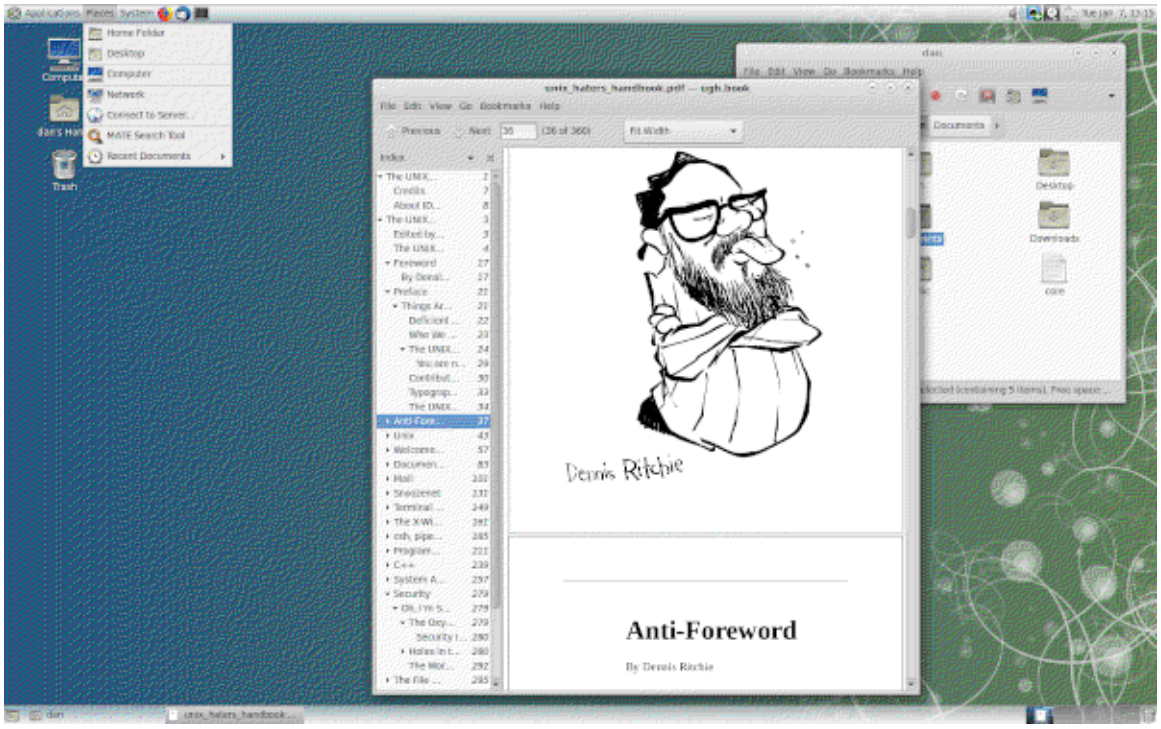
Richard Stallman - the charismatic GNU leader

Although Illumos has imported many Linux utilities, such as `bash` and `vim`, the core utilities are quite different. In terms of complexity the Illumos tools look a lot like simple BSD variants, whereas the GNU utilities bear a closer resemblance to obese dinosaurs. That isn't to say that Solaris is in any way a minimalistic system, on the contrary, it is in many ways a far more professional* system then Linux or the BSD's. The `cmd` directory in the Illumos sources, containing utilities, consists of a staggering 2.5 million

*) read: bloated

lines of code and consists of about 470 distinct projects. In terms of source code, this is about 10 times larger then FreeBSD's collection of utilities. Like the BSD's, and in contrast to Linux, both the Solaris kernel and its userland are written by the same development team, making the system very cohesive and integrated. It was a major player in the enterprise market and backed by serious corporate finance. The SMF service manager, NWAM network manager, and ZFS filesystem makes Systemd, the Linux Network Manager and BtrFS look like a bad joke. FreeBSD has adopted DTrace and ZFS from Solaris, but it's a long way from being as polished and integrated, and FreeBSD Jails cannot hope to compete with Solaris Zones. Today though the corporate backing is gone, and only a thin community remains. Even though they are doing their best to keep the system afloat, innovation has slowed down considerably and the future of Illumos seems bleak at best.

In addition to these issues, Solaris has never been a very good desktop system, however superb it may be as a server. OpenIndiana provides a nice looking Mate desktop, and a handful of the usual user applications, such as Firefox, LibreOffice and VLC. But the repository is very small, in fact in terms of casual desktop usage, even Haiku has a larger selection of applications (with the notable exception that it doesn't have Firefox). There are only three games and five desktops available for example, which compared to BSD and Linux is infinitesimal. The applications that are available may not be very well supported either. On my tests, Wine, VirtualBox and NetBeans were completely broken, and Firefox, Thunderbird and DosBox were somewhat flaky and crash prone. Using Illumos as a desktop will likely involve quite a bit of manual compiling. But here's a pro tip: it's possible to bootstrap NetBSD's pkgsrc on Solaris. Just head over to `www.pkgsrc.org` and read the article about *using pkgsrc on Solaris* (this article talks about Solaris 10, so some details are different in OpenIndiana). Pkgsrc has only about two thirds the Ports of FreeBSD, and to be sure, not all of the packages it does contain will compile easily in Solaris, nevertheless it still means a tremendous expansion of available software. Among other things pkgsrc includes nearly 500 games and over 100 desktops and window managers.



Is it a BSD? Is it a Linux distro? No, it's UNIX man!

The Linux community on the other hand is a very loosey-goosey coalition of developers all around the world, which have created an unbelievable amount of software. It is the only opensource community that has enough breadth to seriously challenge popular commercial offerings. Its open development model has also encouraged research projects, such as CERN and NASA, to use it as their main OS platform. The impact Linux has had on the web and in enterprise is huge, 100% of the top 500 super computers in the world are running Linux, and only two of the top 50 websites in the world are not. BSD, Illumos and other opensource systems have benefited greatly from the popularity of Linux. As most of their applications, developers and users themselves, are imported from its ecosystem. The rise of Linux has proven the viability of opensource systems to the world.

But there are problems in paradise. Even though there are a great *many* different software projects in Linuxland, it is very hard to actually get Linux developers together and work towards a common goal. To some extent this is understandable, since Linux isn't a single project, but many separate projects duck-taped together. The kernel developers have no control whatsoever over the GNU developers, which in turn have no influence over the Mozilla or KDE developers, and so forth. But the fractured nature of Linux goes even deeper then that. As an example, Solaris and FreeBSD lets you choose between two filesystems during installation, the old one, UFS, and the new one, ZFS. In Slackware Linux you are presented with no less then seven filesystems (not including more esoteric options like RaiserFS2 or OpenZFS). All of these are implemented by the same ''group'' of developers. When even the operating systems very core is so fractured, is it then any wonder that applications in Linuxland are divided too? Alas, the promised land floweth with mediocrity and alpha releases! There are literally hundreds of desktops and programming languages available for Linux, but no desktop can rival that of MacOS, and no IDE can rival Virtual Studios.

Other Applications

Graphics		**Multimedia**		**Games**	
name	src	name	src	name	src
feh	14,823	moc	79,496	dosbox*	225,599
gwenview	161,700	audacious	84,771	gnuchess	241,546
gthumb*	366,284	sox*	90,045	nethack*	299,593
imagemagic	654,692	mplayer	456,832	aisleriot	332,703
blender	2,652,234	kdenlive	692,148	hedgewars	604,878
digikam	2,954,228	amarok	1,178,058	supertuxkart	1,054,936
gimp*	3,003,548	ffmpeg*	1,257,600	kde4-games	2,269,842
krita	3,487,103	audacity*	1,472,655	scummvm	3,683,377
inkscape*	2,989,015	vlc*	2,535,834	wesnoth	4,893,311

When observing this bewildering multitude of projects in the Linux community, it is a strange paradox that pockets of ultra conservatism also exits throughout. There is only one kernel for instance, even though Linus Torvalds have requested forks. And though there are alternatives to Systemd, PulseAudio, X and GNU coreutils, virtually no one uses them. Beyond differing wallpapers and icon sets, there is remarkable monotony among the hundreds, if not thousands, of Linux distros out there. Bizarrely the Linux community is often quite hostile towards other operating systems. Many technologies from Solaris and BSD would have greatly augmented Linux, but there is little interest in the community to import them. To some extent this hostility might be contributed to the Free Software Foundations obsession with freedom. It is fine that the FFS wants to share and share alike, but claiming that they have the sole right to define what ''freedom'' means, is taking things a bit far. BSD and Solaris are free systems, they are developed by volunteers and distributed, source code included, with no strings attached. When the FFS still views then as ''tainted'', they have abandoned discussions of fair play, and entered a political, if not a religious, domain. It is telling that the GNU camp will talk your ear off when it comes to philosophies about freedom, but will not have much to say about the technical aspect of development (in the BSD camp it's the other way around).

In truth BSD, Solaris and Linux, among others wouldn't be where they are today without each other. There is no such thing as a ''perfect system'', they are all beautiful in their own unique way. We are all brothers, so lets stop this destructive infighting, lets hold hands in friendship, share a laugh together, and go pitchfork the Windows users already!

Different Linux distributions have their own support forums and wikis, but it's surprisingly rare to see good documentation. A couple of good resources are *The Linux Bible* and *The Linux Command Line and Shell Scripting Bible*, both from Wiley. The Arch Linux wiki is also a very good source of information, even for distributions that has nothing to do with Arch. A good resource for Illumos, although it is a bit dated, is *The OpenSolaris Bible*, also from Wiley.*

6.6. Desktop

name	src	man	name	src
dwm	1956	3	LXQt	1,115,250
ratpoison	13,429	16	CDE*	1,643,270
i3	52,883	6.5	Enlightenment*	1,931,513
notion*	71,048	8	Xfce	3,022,561
awesome*	90,269	5	Mate*	9,284,364
flwm	3591	4.5	GNOME	15,452,992
jwm	27,639	27	Trinity	33,730,011
twm*	33,280	24	KDE4	14,434,682
fluxbox	84,375	2	Plasma	54,140,352
icewm	148,483	18	Compiz*	182,362
wmaker	215,057	3	FLTK	246,333
fvwm2	239,920	142	GTK4	3,069,379
afterstep	247,023	-	Qt5	34,276,808

*) No, I am not sponsored by Wiley, but if you are reading this Wiley - give me a call and we'll talk terms...

There is a huge variety of alternative desktops and window managers for Linux, the FreeBSD Ports Collection has about 100 projects listed under its `x11-wm` category for instance, and the choices in most Linux distros are even greater. Naturally the vast majority of these projects are small window managers, full blown desktop environments are more rare. To simplify immensely, there are two big ones: The GTK-based GNOME and the Qt-based KDE. GNOME3 proved highly controversial, and has inspired numerous spinoffs and rewrites, such as Cinnamon, Unity, Pantheon, Budgie and COSMIC (a Rust rewrite of the latter is in the works), in addition to the old timers GNOME2 (in the form of Mate), and Xfce. The Qt family has some disparity too, albeit not as dramatic; Trinity is a continuation of KDE3, and LXQt and Lumina are lightweight desktops. There are desktop initiatives besides these two families, but interest in them is scarce. Enlightenment, with its EFL toolkit, is probably the most successful, and CDE and GNUStep, although important in the past, has long seen their heyday. Mezzo and Project Looking Glass were examples of truly novel GUI design, but they were totally ignored by the community.

If we look at window managers* the list becomes even larger! You can basically divide these numerous projects into two main categories; stacking window managers that follows a traditional place-and-resize-by-mouse window placement, and keyboard-centric tiling window managers that auto-place windows in a grid. Auto-snap and tiling-mode extensions in modern desktops are highly influenced by the latter. Heated flame wars about which is the ‘‘best’’ desktop is a popular and time honored pass-time for Linux users. And there is no end to online blogs discussing the minutia of GUI differences from one distro to the next. Superficially though, there is no great desktop difference between the distros we have analyzed in this article, save for TinyCore, which provides a very unusual and super minimalistic experience. All the others offer popular desktop alternatives, and let you install a host of more obscure window managers. If fact, there is surprisingly little difference even between the large families of Linux distributions. Sure, there are different package management systems, and front-end admin tools, but at the end of the day, you usually work on the same programs. Any distro can be made to look as familiar or unfamiliar as you like, and tweaked and mangled to suit whatever needs you have. There are heated flame wars to be sure, but in actuality they are usually pointless arguments over trivialities.

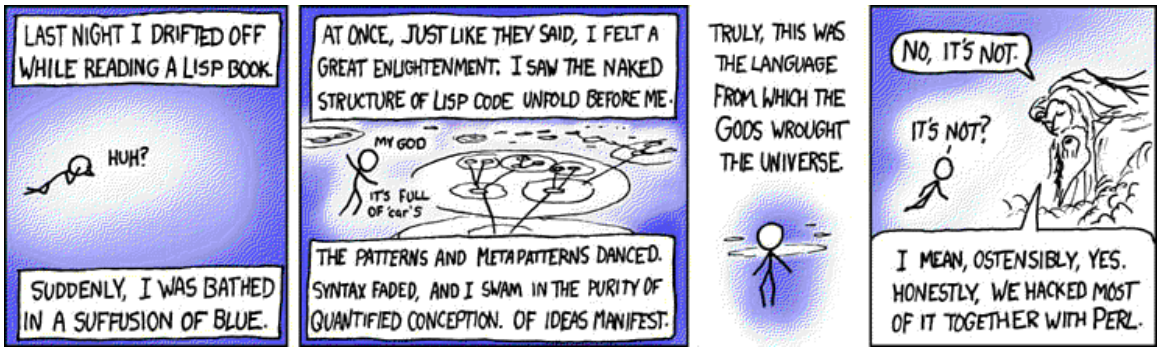
The desktops listed above have a fairly comprehensive count of their respective source code projects, but they do not include everything, and more importantly, not dependencies. If they did, these numbers would easily triple or more. Despite the wealth of choices, no Linux desktop or window manager that I am aware of, do a good job of integrating with the terminal and following UNIX design principals, as the Plan 9 window manager does. One problem here is X itself. X is a 10 million line convoluted monstrosity, building something simple with it would be like creating a well run department within a suffocating bureaucracy. It is an oxymoron. Tiny Core ships with its own fork of the ancient Xvesa version of X. It is not especially pretty, but at 1/100 the size of modern X, the simplicity at least is refreshing.

In recent years the Linux community has been very hyped over the new Wayland display server. It is indeed good that modern UNIX finally gets an alternative to X, but as far as simplicity and UNIX design principals are concerned, Wayland does not, by itself, have much to offer. But it does push much more of the development responsibility onto the individual graphical applications. This is both good and bad. On the one hand, it means that Wayland is simpler then X, and it means that desktops and big graphical applications have more freedom to design the interface they want. On the other hand, it does mean that it requires more effort to develop window managers and simple applications. In theory though, developers have more freedom to experiment with new interfaces, and hopefully this can stimulate more innovative and effective designs. So far though, truly innovative ideas have been scarce, and realistically, Wayland will probably do the very same thing that X did; fuel the drive to copycat attractive but ineffectual proprietary systems.

---

*) A window manager, is a program that does exactly that; manages windows. It might provide a menu, maybe even a panel, but no toolkit, applets, docklets, whatlets and other applications that a full desktop environment provides. As such they are usually orders of magnitude simpler then mainstream alternatives, and they can be highly productive in the hands of an expert, but novice users will frequently find them challenging and/or boring.

6.7. Programming



name	src	name	src	name	src
rust*	12,666,824	lua*	33,600	cvs*	125,645
clang*	10,912,614	sbcl*	531,780	mercurial*	517,670
gcc*	9,642,336	clisp	879,276	git*	849,017
OpenJDK*	9,038,582	tcl*	1,055,095	subversion	1,419,103
go*	1,911,723	perl*	1,298,701	sqlite*	322,030
ghc	660,821	python3*	1,312,803	postgresql*	1,633,084
nasm*	158,257	ruby*	1,950,412	mariadb	4,397,646

Solaris came with its own development suit called Sun Studio, which included compilers for C, C++ and Java, and other basic tools. Both Sun Studio, GCC and Clang can be installed in Illumos. Perl and Python are included by default. Most Linux distros do not ship with many programming utilities, but all of them have a rich set of tools available in their repositories. In fact Linux has become somewhat of a breeding ground for bleeding edge computer science, and most new and weird programming languages are born in this chaotic environment. This versatility is both good and bad, since the great multitude of choices can be very confusing for aspiring new developers.

Another serious issue, that is clearly seen from the numbers above, it the share complexity of modern programming languages. The question of whether it is the complexity of applications that is bloating the languages, or if the languages are bloating the applications, is somewhat of a chicken-or-egg question. In any event, the situation is not pleasant. Recent years has seen a lot of new development in this area, such as the memory-safe languages Rust and Go. It is indeed good that memory safe languages can protect new programs against bugs that have plagued C and C++ programs for the last five decades. Yet, in my humble opinion, these tools only provide a soothing bandaid over an infected wound, they do not solve the underlying problem. Software breaks because the developers do not comprehend their code. In theory there would be no bugs, if programmers foresaw the full consequences of their code. Of course, that is unrealistic under the best of circumstances, but in modern development environments it is down right hopeless. More sophisticated languages and tools can in some ways alleviate the pain, but in other subtle ways, they increase it.

Beyond the usual programming books, the classic *Advanced Programming in the UNIX Environment* is a good resource for Illumos, while *The Linux Programming Interface* is a more suitable alternative on Linux. Of course both of these books talk about deep system programming using syscalls and what not, if you are interested in game, web, or just plain graphical programming, you should look elsewhere. There is a plethora of other programming resources from O'Reilly and other publishers, and as always the internet is overflowing with more or less useless information on this, and any other, subject.

6.8. Kernel

Linus Torvalds - the benevolent Linux dictator

	Linux		**Illumos**	
name	src	name	src	comment
include	693,562	sys	1,085,706	headers
kernel	265,345	os	176,025	kernel facilities
crypto	82,252	crypto	38,108	cryptographic support
security	71,219			various security features
drivers	14,661,237	io	2,670,504	device drivers
fs	1,050,708	fs	446,160	filesystems
net	881,146	inet	242,837	network
arch	1,763,560	arch	692,998	architecture dependent code
misc	2,966,019	misc	139,716	miscellany
sum	7,773,811	sum	2,821,550	sum without drivers
all	22,435,048	all	5,492,054	sum with drivers

The numbers for the Illumos kernel is fairly similar to FreeBSD (except that a lot of driver related stuff is included in the Illumos header files), while the Linux kernel is to-three times bigger or more in every category. There isn't really any good resources for these kernels, primarily because they are just too complex. Many operating system courses use *The Design and Implementation of The FreeBSD Operating System* in its curriculum, and this is probably the best indirect reference book for the Linux and Solaris kernels.

7. Concluding thoughts

EVERYTHING IS TERRIBLE
- Michael W. Lucas

What conclusions can we draw from our study? As stated in the introduction, simplicity and perfection can go too far. A system void of features and practical value is worthless, however ‘‘perfect’’ the code otherwise may be. Finding wisdom, as always, is a question of finding balance. I do not have sufficient insight to give any definitive answer as to where the line between simplicity and functionality should go. And, really, this study has been little more then putting some specific numbers on the complexity of UNIX-like operating systems.

That said, we must not confuse greater quantity for greater quality, or material progress with greater understanding. Although computers have gained materially in many ways, I would argue that the overall understanding and wisdom when it comes to computer science has declined the past 50 years. Though few people used computers in the 70’s, those that did wrote their own compilers, today it’s impressive when an average user can locate his or her own files without needing tech support. And who can blame them, even the developers don’t understand what they are developing. A good example to illustrate this modern phenomena is the aforementioned classic text book, *The UNIX Programming Environment.* This book was published in 1984, and I have yet to see any book about operating systems that can match its clear insights. It’s not the book that’s startling, but the operating system that allowed such a book to be written.

Why have computer science stagnated in the midst of tremendous technological innovations? The answer is simple. You cannot truly advance a system you don’t comprehend. No human, past or present, can study 20 million lines of programming code in any meaningful way. If operating systems are to be understood, and thus be a vehicle for real progress, for the individual as well as for society, they would have to be smaller then our current popular choices by orders of magnitude. No doubt popular distros and apps will continue to grow more features in the years to come, but will they really advance human knowledge? Whereas most of the ancient materials of the Greek culture is eroded away, only scraps of ruins and museum objects left, parts of their culture had lasting benefits for humanity. The works of Pythagoras and Euclid are taught in mathematical classes today, and probably will be taught a 1000 years hence.

But the internet, which we so love and worship today, will be less then dust by then. However, computer science has the potential to benefit humanity in lasting ways. The UNIX authors did uncover some axioms about system design, that will prove true for all time (whether future generations has wisdom enough to heed them, is an altogether different question). And I believe other insights could also be gained, if we can manage to look beyond the tip of our collective noses. The authors of UNIX did not mean for their system to be the end of computer science, but a beginning. And I don’t think UNIX offers the only paradigm worth pursuing. The Lisp machines from the 80’s for instance, offered an interesting ‘‘top-down’’ development model, as opposed to the ‘‘bottom-up’’ approach of UNIX. The old UNIX system can naturally be improved, but it is rare to see wisdom in the numerous attempt at doing so. Plan 9 did advance this fine foundation. And recently, SerenityOS has also made a very promising rewrite. Hopefully, such progress will not go totally unnoticed.

Today though, the ability to study deeply an operating system that you can actually use, is largely lost, and with it, much of the force in computer science. Can it be revived? Sure. In fact, good initiatives already exist. But simplicity is hard, and true progress follows a narrow road. The fact that modern technology has made us conceited and comfortable, does not make it any easier. To quote Albert Einstein: *Man like every other animal is by nature indolent. If nothing spurs him on, then he will hardly think, and will behave from habit like an automation.* Perhaps that is why UNIX produced such glorious minds, not only did it have the sufficient simplicity, but also the right amount of annoyance?

APPENDIX A

Collecting the Statistics

The following sections describe the nitty gritty details of how the statistics in this article were collected. If some of the numbers have been doctored, or if they are mere guesswork, it shall all be revealed here. And yes, simplicity takes precedence over accuracy in my methods. A good example is source code counts; In most cases I have taken the number straight from the summary of `cloc` (a good alternative to this program is `sloccount`). `cloc` does however include many files in the count that aren't actually programming code, such as PO translation files, HTML documentation, and subtle errors in `cloc` itself means that the counts aren't always 100% correct. I have chosen to ignore such issues and leave the numbers as is. The statistic is accurate enough to be informative, and though some applications may have numbers that give an inflated impression of their code complexity (eg. many KDE applications), it nevertheless gives a fair estimate of overall development effort, including to some degree, documentation. Another issue I have chosen to side step is dependencies. Deceptively simple programming code can rely on a huge array of dependencies, but I do not have the resources to go down that rabbit hole. This is only a casual study, and I trust mistakes have been made, but feel free to duplicate and verify my findings.

Note: Tabs are written as \t in order to make them visible.

1. ANCIENT UNIX

The early UNIX systems prior to V5 have largely been lost, but recently the V1 kernel source was recovered. Together with surviving V2 userland it is possible to create a bootable system. For simplicity this system is referred to as ‘‘V1’’ in this document, but in actuality it is a weird V1/V2 jerry rigged hybrid. V9 and V10 of Research UNIX was finally released to the public only a few years ago. There are gaps in the sources however (especially for V9), and it is not currently possible to use V10 practically in an emulator. You can run V9, but it is slightly cumbersome. These editions have never actually been distributed in a complete and installable form, they were more ‘‘conceptual’’ then real releases. As for the commercial System III and V releases, it is possible to get their sources on the internet and run them in emulators, but the legality of doing so is highly suspect. Hopefully these ancient proprietary systems will be opensourced in the future. SIMH emulators were used to run the other ancient systems discussed in this article, the PDP11 emulator for V1, V5, V6 and V7, and the VAX780 emulator for V8 and the BSD's.

Note: commands such as `du` and `ps` report size in blocks of 512 bytes, divide this by 2 to get size in kilobytes. Traditionally this was the filesystem block size on UNIX, but 4BSD doubled this to 1 kilobyte blocks, which more then doubled filesystem performance. Since then the filesystem block size has steadily increased on UNIX systems, but classic utilities will sometimes use the traditional 512 byte ‘‘blocks’’ non the less (although newer systems largely use the more intuitive 1 Kb ‘‘blocks’’).

man

```
sh # for BSD, to switch from csh
for man in ‘ls /usr/man/man[1-8]‘; do # for V8 use man[1-9w]
  man ‘echo $man | sed ’s/\(.*\)\.\(.*\)/\2 \1/’’ | sed ’/^$/d’ | wc -l
done | awk ’{ sum += $1 } END { print sum }’
```

Explanation: This command will count non-empty lines of text in all manpages, divide this by 55 and you have approximately the number of manual pages. Running `man n program` will display at least one full page of text, but most of the manpages in these early days were only a few lines long, so the line count will include mostly empty lines. To avoid this problem the empty lines are pruned off with `sed '/^$/d'` before the lines are counted. Of course this too is slightly inaccurate, but less so.

For V6 the number is just an educated guess based on how much diskspace the manuals require (˜2.7 pages/Kb). For V5 and V1 the manuals can be obtained as a PDF, but as mentioned above most of these pages are just whitespace, so the number given is just a rough estimation of the page count if whitespace was removed. As for V10 manpages see Linux/BSD instructions for hints on how to count them.

bin

```
ls -l /bin /usr/bin /etc | grep 'rw[xs]' | wc -l # for V5/V6 just wc
# for BSD:
ls -l `echo $PATH | sed -e 's/\.//' -e 's/:/ /g'` | grep 'rw[xs]' | wc -l
# for V1:
ls /bin >TEMP; wc TEMP; rm TEMP
ls -l etc # manually count the [ls]xrwr- files
```

The V10 number is the files/directories in `cmd`, `dk/cmd`, `games`, `ipc/bin`, `ipc/internet`, `ipc/perf`, `ipc/servers`, in `/usr/src` and `/usr/jerq/src`.

files

```
du -a / | wc -l # for V5/V6 just wc
# for V1:
check
```

The number of files in the V10 source code can be counted on the host, but this number is obviously lower then what it would be on a real system (maybe about 2/3 of the files on a real system, and even less for V9). As for 4.1 BSD the system I used did not include source code for user applications, and V5 did not include any source code or even manuals, so the numbers given are only partial.

conf

```
ls -l /etc | grep rw- | wc -l # for V5/V6 just wc
# for V1:
ls -l /etc # manually count the s-rwr- files
```

pss, mem

```
ps alx
```

Explanation: Calculating these numbers automatically is problematic, since a command such as `ps alx | wc -l` and `ps alx | awk '{ sum += $10 } END { print sum }'` on V7 would spawn 3 additional processes and gobble up 36 extra kilobytes of memory, a surprisingly big deal back in the day. Therefor the best way to get fair statistics is just to analyze the output of `ps alx` manually. For V5 and V6 the memory usage can be found in column 6, for V8 column 2. For V7 use the SZ column and for the BSD's use the RSS column (size / 2 = Kb). As for V1, it has no program for checking memory usage.

src

```
# V7 (the double loop is necessary because of memory constraints)
cd /usr/src
for dir in * /usr/sys /usr/include; do
  for file in `du -a $dir | awk '/.[chsy]$/ { print $2 }'`; do
    sed -e '/^[ \t]*$/d' -e '/^[ \t]*\*/d' $file | wc -l
  done | awk '{ sum += $1 } END { print sum }'
done | awk '{ sum += $1 } END { print sum }'

# V8
for file in `du -a /usr/sys /usr/src /usr/include | \
  awk '/.[chsy]$/ { print $2 }'`; do
  sed -e '/^[ \t]*$/d' -e '/^[ \t]*\*/d' $file | wc -l
done | awk '{ sum += $1 } END { print sum }'

# BSD 4.1, 4.3 (kernel and headers only)
sed -e '/^[ \t]*$/d' -e '/^[ \t]*\*/d' \
`du -a /sys /usr/include | awk '/.[chsy]$/ { print $2 }'` | wc -l
```

The source code for V6, BSD 4.3 and V10, have all been obtained and counted outside of these systems. You can easily get these files with a quick Google search, and analyze the code with tools such as `cloc` or `sloccount`. The source code for V5 and V1 are educated guesses based on the disk size of `/usr/source` and `/usr/sys` (size * 40 lines/Kb).

hdd

```
du -s / # (size / 2 = Kb)
# for V1:
check
```

2. INFERNO

These instructions assumes you are running hosted Inferno on a Plan 9 system (although running it from Linux shouldn't require any changes), and that you have the following custom commands available in your Inferno environment:

```
fn lsprefix{ du -a | grep -i '\.'$1'$' | awk '{ print $2 }' }
fn sloc{ sed -e '/^[ \t]*$/d' -e '/^[ \t]*#/d' -e '/^[ \t\/]*\*/d' $* | wc -l }
fn awk{
        if { ~ $#* 1 } { file = /fd/0 } { file = $2 }
        os -d $emuroot^'{pwd} awk $1 $file
}
```

man

```
cd /man
for man in '{ls [0-9]*} {
        man '{echo $man | sed 's/\// /'} | wc -l} |
          awk '{ sum += $1 } END { print sum }'
```

Divide this by 55 and you have approximately the number of manual pages.

bin

```
ls /dis/* | wc -l
```

files

```
du -a / | wc -l
```

conf

Inferno has very few configuration files, and no `/etc` equivalent. Exactly what constitute essential system configuration files is a question of definition. But at the very least a sysadmin needs to maintain these three files:

`/lib/ndb/local`	main network configuration
`/lib/wmsetup`	system/desktop startup configuration
`$home/lib/plumbing`	plumber configuration

pss

```
ls /prog | wc -l
```

src

```
for dir in /appl /include /lib* /limbo /module /os /utils {
        cd $dir; for src in '{lsprefix [bchmsy]} { sloc $src } } |
        awk '{ sum += $1 } END { print sum }'
```

Side note about source code: code from `/include`, `/lib*`, `/limbo`, `/os` and `/utils` are used to build the system. The Limbo code for the Inferno applications are in `/appl` and `/module`.

mem

```
wm/task
```

hdd

```
echo '{du -s / | sed 's/ .*//'} / 1024 | calc
```

3. Plan9Port

Plan9Port not an operating system, but a collection of Plan 9 applications ported to UNIX. The instructions assume you are running these applications on a Slackware Linux system, but the basic approach should be the same for any UNIX host.

man

```
9 rc
cd $PLAN9/man
awk '{ sum += $1 } END { print sum }' <{
        for(man in man*/*.[1-9]*){
                man '{echo $man | sed 's/.+/(.+).(.+)/2 1/'}}}
```

Divide this by 55 and you have approximately the number of manual pages.

bin

```
ls $PLAN9/bin | wc -l
```

files

```
du -a $PLAN9 | wc -l
```

conf

Like Plan 9, there are only a few configuration files for Plan9Port. A sysadmin needs to maintain at least three files:

`$PLAN9/ndb/local`	main network configuration
`$HOME/lib/profile`	user startup configuration
`$HOME/lib/plumbing`	plumber configuration

pss

```
9 rc
/bin/ps -ely | awk '/rio|9term|devdraw|acme|9pserve|rc$/
        { print }' | wc -l
```

Explanation: The idea here is to start up a basic Plan9Port desktop environment, and measure the number of Plan9Port processes only, excluding the other processes running on the host. What constitutes a ''basic Plan9Port desktop'' is quite arbitrary of course. I have tested a rio desktop running an acme editor with a win rc shell, and a 9term running a rc shell. It may also be noted that these programs have significant host dependencies, such as X, which are not included in the count.

src

```
cloc $PLAN9     # install cloc on the host
```

mem

```
9 rc
/bin/ps -ely | awk '/rio|9term|devdraw|acme|9pserve|rc$/
        { sum += $8 } END { print sum/1024, "Mb" }' | wc -l
```

See above explanation in the pss section.

hdd

```
/bin/du -hs $PLAN9
```

4. Plan 9

In the following instructions I use these custom commands to simplify some of the examples:

```
fn lsprefix{ du -a | grep -i '\.'$1'$' | awk '{ print $2 }' }
fn sloc{ sed -e '/^[ \t]*$/d' -e '/^[ \t]*#/d' -e '/^[ \t\/]*\*/d' $* | wc -l }
```

man

```
cd /sys/man
awk '{ sum += $1 } END { print sum }' <{
        for(man in '{ls [0-9]*}) {man '{echo $man | sed 's/\// /'} | wc -l}}
```

Divide this by 55 and you have approximately the number of manual pages.

bin

```
ls /bin/* | wc -l
```

files

```
du -a /root | wc -l
```

conf

Plan 9 has few configuration files, and no `/etc` equivalent. Exactly which files can be considered as essential system configuration is a question of definition. But at the very least a sysadmin needs to maintain these five files:

`plan9.ini`	boot configuration
`/lib/ndb/local`	main network configuration
`$home/lib/profile`	user startup configuration
`$home/lib/plumbing`	plumber configuration
`$home/bin/rc/riostart`	desktop startup configuration

pss

```
ls /proc | grep -v trace | wc -l
```

src

```
cd /sys/src
awk '{ sum += $1 } END { print sum }' <{
        for(src in '{lsprefix [chsy]}) sloc $src}
```

pkg

```
9fs 9front
ls /n/extra
9fs 9contrib
ls /n/contrib
9fs 9pio
ls /n/9pio/extra
ls /n/9pio/sources
```

These commands are for 9front, the 9pio repositories are the old Bell Labs sources, many of which will no longer compile. Exactly how many ‘‘packages’’ these resources holds is a question of definition.

As for 9legacy, the old method `9fs sources` no longer work since the Bell Labs server is down. You can add these lines to `/bin/9fs` to get the 9pio and 9front repositories mentioned above. Beware though that 9legacy and 9front are slightly incompatible systems, so many of these packages will not compile.

```
case 9pio
        srv -nq tcp!9p.io 9pio && mount -nC /srv/9pio /n/9pio
case 9front
        9fs 9front.org
        for(i in 9front extra fqa hardware iso lists pkg sites)
                bind /n/9front.org/$i /n/$i
case 9contrib
        9fs contrib.9front.org
        for(i in contrib sources)
                bind /n/contrib.9front.org/$i /n/$i
```

mem

```
memory
```

The numbers used here are a bit doctored. The fileserver in Plan 9 (fossil in 9legacy and cwfs64 or hjfs in 9front) uses a sizable chunk of memory as a cache. However this filesystem cache is usually idle, so after running memory I ran `ps | grep <FILESYSTEM>` and subtracted this cache from the results.

Statistics for memory usage is posted in `/dev/swap` , but this info is not exactly user friendly. 9front comes with a script called `memory` that translates these numbers. 9legacy does not have this script, but it is simple enough to implement it:

```
#!/bin/rc
awk '
function human(name, n) {
        printf "%-15s", name
        if(n >= 1000000000) printf "%.3g GB\n", n / 1073741824
        else if(n >= 1000000) printf "%.3g MB\n", n / 1048576
        else if(n >= 1000) printf "%.3g KB\n", n / 1024
        else printf "%d B\n", n
}
$2 == "memory" { human("total", $1) }
$2 == "pagesize" { pagesize = $1 }
$2 == "kernel" && NF == 2 { human("total kernel", $1 * pagesize) }
$2 == "user" {
        split($1, a, "/")
        human("total user", a[2] * pagesize)
        print ""; human("used  user", a[1] * pagesize)
}
$2 == "kernel" && $3 == "malloc" { split($1, a, "/"); human("used  kernel", a[1]) }
$2 == "kernel" && $3 == "draw" { split($1, a, "/"); human("used  draw", a[1]) }
 ' < /dev/swap
```

hdd

```
du -hs /root
# alternatively for hjfs on 9front:
echo df >> /srv/hjfs.cmd
```

5. Minoca

bin

```
ls $(echo $PATH | sed 's/:/ /g') | wc -l
```

files

```
opkg update; opkg install du
du -a / | wc -l
```

conf

```
du -a /etc | wc -l
```

pss

```
ps
```

src

You can download the Minoca sources from github and analyze it on your host system with `cloc` or a similar tool. PS: Although the ‘‘latest’’ Minoca image has been analyzed here, the operating system hasn’t actually changed at all the last five years, as far as I can see.

pkg

```
opkg update
grep Source /var/opkg-lists/main | wc -l
```

mem

Memory statistics is displayed on top of the screen

hdd

Check the filesize of the downloaded Minoca image

6. SerenityOS

man

```
wc -l /usr/share/man/man*/*.md
```

Divide this by 55 and you have approximately the number of manual pages.

bin

```
ls /bin | wc -l
```

files

```
du -a / | wc -l         # as root
```

conf

```
du -a /etc | wc -l      # as root
```

src

```
du -a /usr/src | cut -f 2 |
  xargs egrep -v '^[ \/]*($|//|\*[ \/]|\*$)' | wc -l
```

We use `cut` and `egrep` here since SerenityOS do not include `sed` or `awk` by default. The result in not entirely accurate; Serenity does not include the sources of the shell Userland Utilities (a mere oversight perhaps?), which would add some additional 20,000 lines to this count.

pkg

```
ls Ports | wc -l        # from the host
```

mem

```
SystemMonitor
```

hdd

```
df -h
```

7. Haiku

man

```
makewhatis /boot/system/documentation/man
cd /boot/system/documentation/man
for man in $(ls man[1-8]); do
        man $(echo $man | sed 's/\(.*\)\.\(.*\)/\2 \1/') | wc -l
done | awk '{ sum += $1 } END { print sum }'
```

Divide this by 55 and you have approximately the number of manual pages.

bin

```
for dir in $(echo $PATH | sed -e 's/^..//' -e 's/:/ /g'); do
        ls $dir
done | wc -l
```

files

```
du -a /boot /dev | wc -l
```

conf

```
du -a /boot/system/settings | wc -l
```

pss

```
ps | wc -l
```

src

You can get the sources by running `git clone https://github.com/haiku/haiku` , and analyze it with a program like `cloc` (It's probably easiest to do this in a different operating system).

pkg

```
HaikuDepot
```

mem

```
ActivityMonitor
```

hdd

It's surprisingly hard to measure this accurately in Haiku, since there are symlinks all over the place, which `df` and `du` are'nt smart enough to realize. In the end I ran `du -h haiku.qcow2` on the host.

8. Minix

man

```
for dir in /usr/man /usr/X11R7/man; do
        cd $dir
        for man in $(ls man[1-9]); do
                man $(echo $man | sed 's/\(.*\)\.\(.*\)/\2 \1/') | wc -l
        done
done | awk '{ sum += $1 } END { print sum }'
```

Divide this by 55 and you have approximately the number of manual pages.

bin

```
for dir in $(echo $PATH | sed 's/:/ /g'); do
        ls $dir
done | wc -l
```

files

```
du -a / | wc -l
```

conf

```
du -a /etc | wc -l
```

pss

```
ps ax | wc -l
```

src

You can get the sources by running `git clone https://github.com/Stichting-MINIX-Research-Foundation/minix` , and analyze it with a program like `cloc` (It's probably easiest to do this in a different operating system). I recommend downloading the latest development snapshot of Minix, and not the stable release. Actually, nothing new has happened in Minix the last five years, and the project seems to be dormant, if not dead.

pkg

```
pkgin update
pkgin available | wc -l
```

mem

```
ps alx | awk 'NR > 1 { sum += $8 } END { print sum/1024, "Mb"}'
```

hdd

```
df -h
```

9. OpenBSD

man

```
for dir in /usr/share/man /usr/X11R6/man; do
        cd $dir
        for man in $(ls man[1-9]); do
                man $(echo $man | sed 's/\(.*\)\.\(.*\)/\2 \1/') | wc -l
        done
done | awk '{ sum += $1 } END { print sum }'
```

Divide this by 55 and you have approximately the number of manual pages.

bin

```
for dir in $(echo $PATH | sed 's/:/ /g'); do
        ls $dir
done | wc -l
```

files

```
du -a / | wc -l         # as root
```

conf

```
du -a /etc | wc -l      # as root
```

pss

```
ps ax | wc -l
```

Wait a few minutes for the kernel to relink before running this.

src

```
cd /tmp
for pkg in src sys ports xenocara; do
        # choose a mirror close to you
        ftp https://ftp.eu.openbsd.org/pub/OpenBSD/7.1/$pkg.tar.gz
done
cd /usr
mkdir xenocara
tar xzf /tmp/ports.tar.gz
cd src
tar xzf /tmp/src.tar.gz
tar xzf /tmp/sys.tar.gz
cd ../xenocara
tar xzf /tmp/xenocara.tar.gz
pkg_add cloc
cloc /usr/src /usr/xenocara /usr/ports
```

pkg

```
ls /usr/ports/[a-z]* | wc -l
```

mem

```
ps alx | awk 'NR > 1 { sum += $8 } END { print sum/1024, "Mb"}'
```

Wait a few minutes for the kernel to relink before running this.

hdd

```
df -h
```

kernel

I have divided the kernel source into these somewhat arbitrary categories:

`boot`	stand
`sys`	sys
`kern`	kern, lib/libkern
`ddb`	ddb
`compat`	-
`crypto`	crypto
`security`	-
`dev`	dev
`fs`	*fs, uvm
`net`	net*
`misc`	conf, lib(-libkern), scsi
`arch`	arch

10. NetBSD

man

```
for dir in /usr/share/man /usr/X11R7/man; do
        cd $dir
        for man in $(ls man[1-9]); do
                man $(echo $man | sed 's/\(.*\)\.\(.*\)/\2 \1/') | wc -l
        done
done | awk '{ sum += $1 } END { print sum }'
```

Divide this by 55 and you have approximately the number of manual pages.

bin

```
# as root
for dir in $(echo $PATH | sed 's/:/ /g'); do
        ls $dir
done | wc -l
```

files

```
du -a / | wc -l         # as root
```

conf

```
du -a /etc | wc -l      # as root
```

pss

```
ps ax | wc -l
```

src

```
cd /tmp
for pkg in gnusrc sharesrc src syssrc xsrc; do
        # choose a mirror close to you
        ftp http://ftp.fr.netbsd.org/pub/NetBSD/NetBSD-9.3/source/sets/$pkg.tgz
done
cd /
tar xzf /tmp/*.tgz
pkgin cloc
cloc /usr/src /usr/xsrc
```

pkg

```
ls /usr/pkgsrc/[a-z]* | wc -l
```

mem

```
ps alx | awk 'NR > 1 { sum += $8 } END { print sum/1024, "Mb"}'
```

hdd

```
df -h
```

kernel

I have divided the kernel source into these somewhat arbitrary categories:

`boot`	stand
`sys`	sys
`kern`	kern, lib/libkern
`ddb`	ddb, gdbscripts
`compat`	compat
`crypto`	crypto, opencrypto
`security`	secmodel
`dev`	dev, external/bsd/drm*, modules
`fs`	*fs, uvm
`net`	net*
`misc`	altq, coda, conf, dist, external(-bsd/drm*), lib(-libkern), rump
`arch`	arch

11. DragonFly BSD

man

```
sh # to switch from csh
cd /usr/share/man
for man in $(ls man[1-9]); do
        man $(echo $man | sed 's/\([^.]*\)\.\([^.]*\)\..*/\2 \1/') | wc -l
done | awk '{ sum += $1 } END { print sum }'
```

Divide this by 55 and you have approximately the number of manual pages.

bin

```
sh # to switch from csh
for dir in $(echo $PATH | sed 's/:/ /g'); do
        ls $dir
done | wc -l
```

files

```
du -a / | wc -l         # as root
```

conf

```
du -a /etc | wc -l      # as root
```

pss

```
ps ax | wc -l
```

src

```
cd /usr
make src-create         # as root
pkg install cloc
cloc /usr/src
```

pkg

```
cd /usr
make dports-create-shallow      # as root
ls /usr/dports/[a-z]* | wc -l
```

mem

```
ps alx | awk 'NR > 1 { sum += $8 } END { print sum/1024, "Mb"}'
```

hdd

```
df -h
```

kernel

I have divided the kernel source into these somewhat arbitrary categories:

`boot`	platform/vkernel64
`sys`	sys
`kern`	kern, libkern
`ddb`	ddb
`compat`	-
`crypto`	crypto, opencrypto
`security`	-
`dev`	dev, contrib/dev
`fs`	vfs, vm
`net`	net*
`misc`	bus, compile, conf*, contrib(-dev), gnu, libiconv, libprop, tools
`arch`	cpu, platform/pc64

12. FreeBSD

man

```
for dir in /usr/share/man /usr/share/openssl/man; do
        cd $dir
        for man in $(ls man[1-9]); do
                man $(echo $man |\
                sed 's/\([^.]*\)\.\([^.]*\)\..*/\2 \1/') | wc -l
        done
done | awk '{ sum += $1 } END { print sum }'
```

Divide this by 55 and you have approximately the number of manual pages. A couple of manpages will complain that you need to install `groff` to render them, but doing so doesn't really alter the result.

bin

```
for dir in $(echo $PATH | sed 's/:/ /g'); do
        ls $dir
done | wc -l
```

files

```
du -a / | wc -l         # as root
```

conf

```
du -a /etc | wc -l      # as root
```

pss

```
ps ax | wc -l
```

src

```
pkg install cloc
cloc /usr/src
```

pkg

```
ls /usr/ports/[a-z]* | wc -l
```

mem

```
ps alx | awk 'NR > 1 { sum += $8 } END { print sum/1024, "Mb"}'
```

hdd

```
df -h
```

kernel

I have divided the kernel source into these somewhat arbitrary categories:

`boot`	cddl/boot
`sys`	sys
`kern`	kern, libkern
`ddb`	ddb, gdb, tests, cddl/dev
`compat`	compat, cddl(-boot, dev)
`crypto`	crypto, opencrypto
`security`	security
`dev`	dev, contrib/dev, modules
`fs`	fs, nfs*, ufs, geom, vm, contrib/openzfs
`net`	net*, nlm, rpc
`misc`	bsm, cam, conf, contrib(-dev, openzfs), dts, gnu, isa kgssapi, ofed, teken, tools, xdr, xen
`arch`	amd64, arm*, i386, mips, powerpc, riscv, x86

13. OpenIndiana and OmniOSce

man

```
# for OpenIndiana, rebuild whatis db first
for dir in $(find /usr -type d -name man); do
        MANPATH=$MANPATH:$dir
done
export MANPATH
man -w

for dir in $(find /usr -type d -name man); do
        cd $dir
        for man in $(ls man[1-9]*); do
                man $(echo $man | sed 's/\(.*\)\.\(.*\)/\2 \1/') | wc -l
        done
done | awk '{ sum += $1 } END { print sum }'
```

Divide this by 55 and you have approximately the number of manual pages.

bin

```
for dir in $(echo $PATH | sed 's/:/ /g'); do
        ls $dir
done | wc -l
```

files

```
du -a / | wc -l         # as root
```

conf

```
du -a /etc | wc -l      # as root
```

pss

```
ps -ely | wc -l
```

src

Getting and counting the full source code is a bit tricky. What I have done here is cloned the illumos-gate repository with `git`, which contains the kernel and core userland of Illumos, then run `cloc` on it. This has to be done on a different operating system since OpenIndiana and OmniOS do not have `cloc` in their repositories.

In addition to these sources, most Illumos distributions also bundle their own extra packages, which have not been included in the count. For some, such as OmniOS, these extras are quite small, but for others, such as OpenIndiana, these extras are huge (just adding Firefox and all its dependencies alone will likely multiply the source code ten fold).

pkg

```
# for OpenIndiana
firefox http://pkg.openindiana.org/hipster
firefox http://pkg.openindiana.org/hipster-encumbered
firefox http://sfe.opencsw.org/localhostoih

# for OmniOSce
firefox https://pkg.omniosce.org/r151042/core
firefox https://pkg.omniosce.org/r151042/extra
firefox http://sfe.opencsw.org/localhostomnios
```

mem

```
prstat -t
```

hdd

```
df -h
```

kernel

I have divided the kernel source into these somewhat arbitrary categories: The kernel source can be found in the `/usr/src/uts` category of the illumos-gate repository. Everything here except `common` is added to the `arch` category. And everything in the `common` directory is added to the `misc` category, with the following exceptions:

`sys`	sys
`os`	os, syscall
`crypto`	crypto
`io`	io
`fs`	*fs, vm
`net`	*net*, smb*, rpc*

14. Linux

man

```
# for Debian
cd /usr/share/man
for man in $(ls man[1-8]); do
        man $(echo $man |\
        sed 's/\([^.]*\)\.\([^.]*\)\..*/\2 \1/') | wc -l
done | awk '{ sum += $1 } END { print sum }'

# for openSUSE and Slackware
# for AlmaLinux just drop /usr/local/man here
for dir in /usr/share/man /usr/local/man; do
        cd $dir
        for man in $(ls man[1-9]*); do
                man $(echo $man |\
                sed 's/\([^.]*\)\.\([^.]*\)\..*/\2 \1/') | wc -l
        done
done | awk '{ sum += $1 } END { print sum }'
```

Divide this by 55 and you have approximately the number of manual pages.

Tiny Core and Alpine does not include any manpages by default, but you can manually retrieve the manpages for BusyBox, FLTK, `flwm`, `aterm`, `wbar` (tinyX and fltk_projects do not have manpages), the individual packages that make up the Tiny Core distribution, and ‘‘The Linux man-pages project’’, which documents the Linux kernel. But there is little point in doing so. From a practical point of view, it is far better to just read ‘‘The CoreBook’’ provided by the Tiny Core team. Besides, from a statistical point of view, just counting the Linux kernel man-pages and ignoring the rest will be accurate enough. It’s much the same story for Alpine; the distro just glues BusyBox and musl around the Linux kernel, there is little else.

bin

```
for dir in $(echo $PATH | sed 's/:/ /g'); do
        ls $dir
done | wc -l            # as root for openSUSE and Slackware

# for Debian
for dir in /sbin /usr/sbin $(echo $PATH | sed 's/:/ /g'); do
        ls $dir
done | wc -l
```

files

```
du -a / | wc -l         # as root
```

conf

```
du -a /etc | wc -l      # as root
```

pss

```
ps -ely | wc -l
ps | wc -l              # for Tiny Core and Alpine
```

src

Most Linux distributions makes it easy to inspect the source for individual packages, but to get the entire source code for the install distribution is not that easy. For Slackware you can download the source from one of its many FTP mirrors, and then extract all the archives:

```
lftp -c 'open ftp://ftpmirror.infania.net/slackware;\
mirror -c -e slackware64-15.0/source'
# see: https://mirrors.slackware.com/mirrorlist
find . -name *.t*z -execdir tar xf '{}' ';'
find . -name *.t*bz2 -execdir tar xjf '{}' ';'
```

You can now inspect the code with `cloc`. (these sources are *huge* and you will likely need to analyze them in stages) These figures should correspond fairly well to a generally fleshed out Linux desktop, or any fleshed out UNIX desktop for that matter.

It is also possible to get the Red Hat sources that AlmaLinux is based on, and analyze them in much the same way:

```
wget -r --no-parent https://cdn-ubi.redhat.com/content/public/\
ubi/dist/ubi9/9/x86_64/baseos/source/SRPMS/Packages/a
mv cdn-ubi.redhat.com/content/public/ubi/dist/ubi9/9/x86_64/\
baseos/source/SRPM/Packages Packages
rm -rf cdn-ubi.redhat.com
cd Packages
for dir in [a-z]; do
        cd $dir
        for rpm in *.rpm; do
                rpm2cpio $rpm | cpio -idmv
        done
        cd ..
done
find . -name *.t*z -execdir tar xf '{}' ';'
find . -name *.t*bz2 -execdir tar xjf '{}' ';'
```

Again, the code base here is very large, so you will likely need to analyze it in chunks. Note also that this only copies the BaseOS repository of Red Hat. Typically a Red Hat installation will contain a great number of packages from the AppStream repository as well.

For Tiny Core and Alpine, you can download the source code for the individual packages that make up these distributions, and analyze them (see the above comments in the man section). Although, just counting the lines of code in the Linux kernel and ignoring the rest, will be accurate enough.

pkg

```
apt-cache search . | wc -l      # for Debian
yum list available | wc -l      # for AlmaLinux
zypper packages | wc -l         # for openSUSE
tce-ab                          # for Tiny Core (search for ".")
vi /etc/apk/repositories        # for Alpine
apk update                      # (uncomment community repo first)

# for Slackware, as root
wget https://github.com/sbopkg/sbopkg/releases/download/0.38.1/\
        sbopkg-0.38.1-noarch-1_wss.tgz
installpkg sbopkg-0.38.1-noarch-1_wss.tgz
sbopkg  # sync
ls /var/lib/sbopkg/SBo/15.0/[a-z]* | grep asc | wc -l
```

mem

```
free -m
```

hdd

```
df -h
```

kernel

The source for the categories `include`, `kernel`, `crypto`, `security`, `drivers`, `fs`, `net` and `arch`, can be found in those directories. Everything else I have added to the `misc` category. Note that the `kernel` and `misc` categories are slightly misleading, as the Linux kernel has moved some of the traditional kernel facilities outside of the main kernel directory source.

APPENDIX B

Echo source code

You can endlessly debate the pros and cons of different operating systems, but at the end of the day it's the actual code that tells the story. In this appendix we will list the source code for the `echo` command on various systems. Its a good candidate since it's a very simple program with a clearly defined purpose, the code should reflect this.

1. UNIX V6

```
main(argc, argv)
int argc;
char *argv[];
{
        int i;

        argc--;
        for(i=1; i<=argc; i++)
                printf("%s%c", argv[i], i==argc? '\n': ' ');
}
```

2. UNIX V7, 4.1 and 4.3 BSD

```
#include <stdio.h>

main(argc, argv)
int argc;
char *argv[];
{
        register int i, nflg;

        nflg = 0;
        if(argc > 1 && argv[1][0] == '-' && argv[1][1] == 'n') {
                nflg++;
                argc--;
                argv++;
        }
        for(i=1; i<argc; i++) {
                fputs(argv[i], stdout);
                if (i < argc-1)
                        putchar(' ');
        }
        if(nflg == 0)
                putchar('\n');
        exit(0);
}
```

3. UNIX V8 and V10

V10 adds the special character ’v’, taking up 3 extra lines of code, but is otherwise exactly the same.

```
#include <stdio.h>

main(argc, argv)
char **argv;
{
        register char   *cp;
        register int    i, wd;
        int     j;
        int nflg = 0;
        int escflg = 0;

        while (argc > 1) {
                if (strcmp(argv[1], "-n")==0) {
                        nflg++;
                        argc--;
                        argv++;
                } else if (strcmp(argv[1], "-e")==0) {
                        escflg++;
                        argc--;
                        argv++;
                } else if (strcmp(argv[1], "-ne")==0 ||
                           strcmp(argv[1], "-en")==0) {
                        escflg++;
                        nflg++;
                        argc--;
                        argv++;
                        break;
                } else
                        break;
        }
        for (i = 1; i < argc; i++) {
                for (cp = argv[i]; *cp; cp++) {
                        if (*cp == ’\\’ && escflg)
                        switch (*++cp) {
                                case ’b’:
                                        putchar(’\b’);
                                        continue;
                                case ’c’
                                        return 0;
                                case ’f’
                                        putchar(’\f’);
                                        continue;
                                case ’n’
                                        putchar(’\n’);
                                        continue;
                                case ’r’
                                        putchar(’\r’);
                                        continue;
                                case ’t’
                                        putchar(’\t’);
                                        continue;
                                case ’\\’
                                        putchar(’\\’);
                                        continue;
```

```
                                case '0': case '1': case '2': case '3':
                                case '4': case '5': case '6': case '7':
                                        wd = *cp&07;
                                        j = 0;
                                        while (*++cp>='0' && *cp<='7' && ++j<3){
                                                wd <<= 3;
                                                wd |= (*cp - '0');
                                        }
                                        putchar(wd);
                                        --cp;
                                        continue;

                                default:
                                        cp--;
                        }
                        putchar(*cp);
                }
                if (i < arg-1)
                        putchar(' ');
        }
        if (!nflg)
                putchar('\n');
        return 0;
}
```

4. Plan 9

```
#include <u.h>
#include <libc.h>

void
main(int argc, char *argv[])
{
        int nflag;
        int i, len;
        char *buf, *p;

        nflag = 0;
        if(argc > 1 && strcmp(argv[1], "-n") == 0)
                nflag = 1;

        len = 1;
        for(i = 1+nflag; i < argc; i++)
                len += strlen(argv[i])+1;

        buf = malloc(len);
        if(buf == 0)
                exits("no memory");

        p = buf;
        for(i = 1+nflag; i < argc; i++){
                strcpy(p, argv[i]);
                p += strlen(p);
                if(i < argc-1)
                        *p++ = ' ';
        }

        if(!nflag)
                *p++ = '\n';

        if(write(1, buf, p-buf) < 0){
                fprint(2, "echo: write error: %r\n");
                exits("write error");
        }

        exits((char *)0);
}
```

5. Inferno

```
implement Echo;

include "sys.m";
        sys: Sys;
include "draw.m";

Echo: module
{
        init:   fn(nil: ref Draw->Context, nil: list of string);
};

init(nil: ref Draw->Context, args: list of string)
{
        sys = load Sys Sys->PATH;
        if(args != nil)
                args = tl args;
        addnl := 1;
        if(args != nil && (hd args == "-n" || hd args == "--")) {
                if(hd args == "-n")
                        addnl = 0;
                args = tl args;
        }
        s := "";
        if(args != nil) {
                s = hd args;
                while((args = tl args) != nil)
                        s += " " + hd args;
        }
        if(addnl)
                s[len s] = '\n';
        a := array of byte s;
        if(sys->write(sys->fildes(1), a, len a) < 0){
                sys->fprint(sys->fildes(2), "echo: write error: %r\n");
                raise "fail:write error";
        }
}
```

6. Minoca

Comments in the Minoca source code are extremely verbose, they are not included here.

```
#include <minoca/lib/types.h>

#include <stdio.h>
#include <string.h>

BOOL
EchoIsStringBackslashEscaped (
    PSTR String
    );

INT
EchoMain (
    INT ArgumentCount,
    CHAR **Arguments
    )

{

    PSTR Argument;
    ULONG ArgumentIndex;
    ULONG ArgumentLength;
    CHAR Character;
    ULONG CharacterIndex;
    CHAR DigitCount;
    BOOL EscapeProcessing;
    BOOL PrintTrailingNewline;
    CHAR Value;
    BOOL WasBackslash;

    EscapeProcessing = FALSE;
    PrintTrailingNewline = TRUE;

    for (ArgumentIndex = 1; ArgumentIndex < ArgumentCount; ArgumentIndex += 1) {
        Argument = Arguments[ArgumentIndex];
        if (Argument[0] != '-') {
            break;
        }

        while (TRUE) {
            Argument += 1;
            if (Argument[0] == '\0') {
                break;

            } else if (Argument[0] == 'e') {
                EscapeProcessing = TRUE;

            } else if (Argument[0] == 'E') {
                EscapeProcessing = FALSE;

            } else if (Argument[0] == 'n') {
                PrintTrailingNewline = FALSE;

            } else {
                break;
            }
```

```
    }

    if (Argument[0] != '\0') {
        break;
    }
}

while (ArgumentIndex < ArgumentCount) {
    Argument = Arguments[ArgumentIndex];
    ArgumentIndex += 1;

    if ((EscapeProcessing == FALSE) ||
        (EchoIsStringBackslashEscaped(Argument) == FALSE)) {

        printf("%s", Argument);

    } else {
        Value = 0;
        DigitCount = 0;
        WasBackslash = FALSE;
        ArgumentLength = strlen(Argument);
        for (CharacterIndex = 0;
             CharacterIndex < ArgumentLength;
             CharacterIndex += 1) {

            Character = Argument[CharacterIndex];

            if (DigitCount != 0) {
                if ((Character >= '0') && (Character <= '7')) {
                    Value = (Value * 8) + (Character - '0');
                    DigitCount += 1;
                    if (DigitCount == 4) {
                        DigitCount = 0;
                        printf("%c", Value);
                    }

                    continue;

                } else {
                    DigitCount = 0;
                    printf("%c", Value);
                }
            }

            if (WasBackslash != FALSE) {
                if (Character == 'a') {

                } else if (Character == 'b') {
                    printf("\b");

                } else if (Character == 'c') {
                    PrintTrailingNewline = FALSE;
                    goto MainEnd;

                } else if (Character == 'f') {
                    printf("\f");

                } else if (Character == 'n') {
                    printf("\n");
```

```
                    } else if (Character == 'r') {
                        printf("\r");

                    } else if (Character == 't') {
                        printf("\t");

                    } else if (Character == '\\') {
                        printf("\\");

                    } else if (Character == '0') {
                        Value = 0;
                        DigitCount = 1;

                    } else {

                        printf("\\%c", Character);
                    }

                } else if (Character != '\\') {
                    printf("%c", Character);
                }

                if (Character == '\\') {
                    WasBackslash = !WasBackslash;

                } else {
                    WasBackslash = FALSE;
                }
            }
        }

        if (ArgumentIndex != ArgumentCount) {
            printf(" ");
        }
    }

MainEnd:
    if (PrintTrailingNewline != FALSE) {
        printf("\n");
    }

    return 0;
}

BOOL
EchoIsStringBackslashEscaped {
        PSTR String
        }

{

    if (strchr(String, '\\') != NULL) {
        return TRUE;
    }

    return FALSE;
}
```

7. SerenityOS

```
#include <AK/CharacterTypes.h>
#include <AK/GenericLexer.h>
#include <LibCore/ArgsParser.h>
#include <LibCore/System.h>
#include <LibMain/Main.h>
#include <stdio.h>
#include <unistd.h>

static u8 parse_octal_number(GenericLexer& lexer)
{
    u32 value = 0;
    for (size_t count = 0; count < 3; ++count) {
        auto c = lexer.peek();
        if (!(c >= '0' && c <= '7'))
            break;
        value = value * 8 + (c - '0');
        lexer.consume();
    }
    clamp(value, 0, 255);
    return value;
}

static Optional<u8> parse_hex_number(GenericLexer& lexer)
{
    u8 value = 0;
    for (size_t count = 0; count < 2; ++count) {
        auto c = lexer.peek();
        if (!is_ascii_hex_digit(c))
            return {};
        value = value * 16 + parse_ascii_hex_digit(c);
        lexer.consume();
    }
    return value;
}

static String interpret_backslash_escapes(StringView string, bool& no_trailing_newline)
{
    static constexpr auto escape_map = "a\ab\be\ef\fn\nr\rt\tv\v"sv;
    static constexpr auto unescaped_chars = "\a\b\e\f\n\r\t\v\\"sv;

    StringBuilder builder;
    GenericLexer lexer { string };

    while (!lexer.is_eof()) {
        auto this_index = lexer.tell();
        auto this_char = lexer.consume();
        if (this_char == '\\') {
            if (lexer.is_eof()) {
                builder.append('\\');
                break;
            }
            auto next_char = lexer.peek();
            if (next_char == 'c') {
                no_trailing_newline = true;
                break;
            }
            if (next_char == '0') {
```

```
                lexer.consume();
                auto octal_number = parse_octal_number(lexer);
                builder.append(octal_number);
            } else if (next_char == 'x') {
                lexer.consume();
                auto maybe_hex_number = parse_hex_number(lexer);
                if (!maybe_hex_number.has_value()) {
                    auto bad_substring = string.substring_view(this_index, lexer.tell() \
                    - this_index);
                    builder.append(bad_substring);
                } else {
                    builder.append(maybe_hex_number.release_value());
                }
            } else if (next_char == 'u') {
                lexer.retreat();
                auto maybe_code_point = lexer.consume_escaped_code_point();
                if (maybe_code_point.is_error()) {
                    auto bad_substring = string.substring_view(this_index, lexer.tell() \
                    - this_index);
                    builder.append(bad_substring);
                } else {
                    builder.append_code_point(maybe_code_point.release_value());
                }
            } else {
                lexer.retreat();
                auto consumed_char = lexer.consume_escaped_character('\\', escape_map);
                if (!unescaped_chars.contains(consumed_char))
                    builder.append('\\');
                builder.append(consumed_char);
            }
        } else {
            builder.append(this_char);
        }
    }

    return builder.build();
}

ErrorOr<int> serenity_main(Main::Arguments arguments)
{
    TRY(Core::System::pledge("stdio"));

    Vector<String> text;
    bool no_trailing_newline = false;
    bool should_interpret_backslash_escapes = false;

    Core::ArgsParser args_parser;
    args_parser.add_option(no_trailing_newline, "Do not output a trailing newline", \
    nullptr, 'n');
    args_parser.add_option(should_interpret_backslash_escapes, \
    "Interpret backslash escapes", nullptr, 'e');
    args_parser.add_positional_argument(text, "Text to print out", "text", \
    Core::ArgsParser::Required::No);
    args_parser.set_stop_on_first_non_option(true);
    args_parser.parse(arguments);

    if (text.is_empty()) {
        if (!no_trailing_newline)
            outln();
```

```
        return 0;
    }

    auto output = String::join(' ', text);
    if (should_interpret_backslash_escapes)
        output = interpret_backslash_escapes(output, no_trailing_newline);
    out("{}", output);
    if (!no_trailing_newline)
        outln();
    return 0;
}
```

8. OpenBSD

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <err.h>

/* ARGSUSED */
int
main(int argc, char *argv[])
{
        int nflag;

        if (pledge("stdio", NULL) == -1)
                err(1, "pledge");

        /* This utility may NOT do getopt(3) option parsing. */
        if (*++argv && !strcmp(*argv, "-n")) {
                ++argv;
                nflag = 1;
        }
        else
                nflag = 0;

        while (*argv) {
                (void)fputs(*argv, stdout);
                if (*++argv)
                        putchar(' ');
        }
        if (!nflag)
                putchar('\n');

        return 0;
}
```

9. NetBSD and Minix

```
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* ARGSUSED */
int
main(int argc, char *argv[])
{
        int nflag;

        setprogname(argv[0]);
        (void)setlocale(LC_ALL, "");

        /* This utility may NOT do getopt(3) option parsing. */
        if (*++argv && !strcmp(*argv, "-n")) {
                ++argv;
                nflag = 1;
        }
        else
                nflag = 0;

        while (*argv) {
                (void)printf("%s", *argv);
                if (*++argv)
                        (void)putchar(' ');
        }
        if (nflag == 0)
                (void)putchar('\n');
        fflush(stdout);
        if (ferror(stdout))
                exit(1);
        exit(0);
        /* NOTREACHED */
}
```

10. FreeBSD and DragonFly BSD

```
#include <sys/types.h>
#include <sys/uio.h>

#include <assert.h>
#include <capsicum_helpers.h>
#include <err.h>
#include <errno.h>
#include <limits.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

/*
 * Report an error and exit.
 * Use it instead of err(3) to avoid linking-in stdio.
 */
static __dead2 void
errexit(const char *prog, const char *reason)
{
        char *errstr = strerror(errno);
        write(STDERR_FILENO, prog, strlen(prog));
        write(STDERR_FILENO, ": ", 2);
        write(STDERR_FILENO, reason, strlen(reason));
        write(STDERR_FILENO, ": ", 2);
        write(STDERR_FILENO, errstr, strlen(errstr));
        write(STDERR_FILENO, "\n", 1);
        exit(1);
}

int
main(int argc, char *argv[])
{
        int nflag;      /* if not set, output a trailing newline. */
        int veclen;     /* number of writev arguments. */
        struct iovec *iov, *vp; /* Elements to write, current element. */
        char space[] = " ";
        char newline[] = "\n";
        char *progname = argv[0];

        if (caph_limit_stdio() < 0 || caph_enter() < 0)
                err(1, "capsicum");

        /* This utility may NOT do getopt(3) option parsing. */
        if (*++argv && !strcmp(*argv, "-n")) {
                ++argv;
                --argc;
                nflag = 1;
        } else
                nflag = 0;

        veclen = (argc >= 2) ? (argc - 2) * 2 + 1 : 0;

        if ((vp = iov = malloc((veclen + 1) * sizeof(struct iovec))) == NULL)
                errexit(progname, "malloc");

        while (argv[0] != NULL) {
                size_t len;
```

```
                len = strlen(argv[0]);

                /*
                 * If the next argument is NULL then this is this
                 * the last argument, therefore we need to check
                 * for a trailing \c.
                 */
                if (argv[1] == NULL) {
                        /* is there room for a '\c' and is there one? */
                        if (len >= 2 &&
                            argv[0][len - 2] == '\\' &&
                            argv[0][len - 1] == 'c') {
                                /* chop it and set the no-newline flag. */
                                len -= 2;
                                nflag = 1;
                        }
                }
                vp->iov_base = *argv;
                vp++->iov_len = len;
                if (*++argv) {
                        vp->iov_base = space;
                        vp++->iov_len = 1;
                }
        }
        if (!nflag) {
                veclen++;
                vp->iov_base = newline;
                vp++->iov_len = 1;
        }
        /* assert(veclen == (vp - iov)); */
        while (veclen) {
                int nwrite;

                nwrite = (veclen > IOV_MAX) ? IOV_MAX : veclen;
                if (writev(STDOUT_FILENO, iov, nwrite) == -1)
                        errexit(progname, "write");
                iov += nwrite;
                veclen -= nwrite;
        }
        return 0;
}
```

11. Illumos (ei. Solaris)

```
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <string.h>
#include <locale.h>

int
main(int argc, char *argv[])
{

        register char   *cp;
        register int    i, wd;
        int     j;
        wchar_t         wc;
        int             b_len;
        char            *ep;

        (void) setlocale(LC_ALL, "");

        if (--argc == 0) {
                (void) putchar('\n');
                if (fflush(stdout) != 0)
                        return (1);
                return (0);
        }

        for (i = 1; i <= argc; i++) {
                for (cp = argv[i], ep = cp + (int)strlen(cp);
                    cp < ep; cp += b_len) {
                    if ((b_len = mbtowc(&wc, cp, MB_CUR_MAX)) <= 0) {
                            (void) putchar(*cp);
                            b_len = 1;
                            continue;
                    }

                    if (wc != '\\') {
                            (void) putwchar(wc);
                            continue;
                    }

                    cp += b_len;
                    b_len = 1;
                    switch (*cp) {
                            case 'a':       /* alert - XCU4 */
                                    (void) putchar('\a');
                                    continue;

                            case 'b':
                                    (void) putchar('\b');
                                    continue;

                            case 'c':
                                    if (fflush(stdout) != 0)
                                            return (1);
                                    return (0);

                            case 'f':
```

```
                                        (void) putchar('\f');
                                        continue;

                                case 'n':
                                        (void) putchar('\n');
                                        continue;

                                case 'r':
                                        (void) putchar('\r');
                                        continue;

                                case 't':
                                        (void) putchar('\t');
                                        continue;

                                case 'v':
                                        (void) putchar('\v');
                                        continue;

                                case '\\':
                                        (void) putchar('\\');
                                        continue;
                                case '0':
                                        j = wd = 0;
                                        while ((*++cp >= '0' && *cp <= '7') &&
                                            j++ < 3) {
                                                wd <<= 3;
                                                wd |= (*cp - '0');
                                        }
                                        (void) putchar(wd);
                                        --cp;
                                        continue;

                                default:
                                        cp--;
                                        (void) putchar(*cp);
                                }
                        }
                        (void) putchar(i == argc? '\n': ' ');
                        if (fflush(stdout) != 0)
                                return (1);
                }
                return (0);
}
```

12. GNU (Linux and Haiku)

```
#include <config.h>
#include <stdio.h>
#include <assert.h>
#include <sys/types.h>
#include "system.h"

/* The official name of this program (e.g., no ‘g’ prefix).  */
#define PROGRAM_NAME "echo"

#define AUTHORS \
  proper_name ("Brian Fox"), \
  proper_name ("Chet Ramey")

/* If true, interpret backslash escapes by default.  */
#ifndef DEFAULT_ECHO_TO_XPG
enum { DEFAULT_ECHO_TO_XPG = false };
#endif

void
usage (int status)
{
  /* STATUS should always be EXIT_SUCCESS (unlike in most other
     utilities which would call emit_try_help otherwise).  */
  assert (status == EXIT_SUCCESS);

  printf (_("\
Usage: %s [SHORT-OPTION]... [STRING]...\n\
  or:  %s LONG-OPTION\n\
"), program_name, program_name);
  fputs (_("Echo the STRING(s) to standard output.\n\
\n\
  -n             do not output the trailing newline\n\
"), stdout);
  fputs (_(DEFAULT_ECHO_TO_XPG
           ? N_("\
  -e             enable interpretation of backslash escapes (default)\n\
  -E             disable interpretation of backslash escapes\n")
           : N_("\
  -e             enable interpretation of backslash escapes\n\
  -E             disable interpretation of backslash escapes (default)\n")),
         stdout);
  fputs (HELP_OPTION_DESCRIPTION, stdout);
  fputs (VERSION_OPTION_DESCRIPTION, stdout);
  fputs (_("\
\n\
If -e is in effect, the following sequences are recognized:\n\
"), stdout);
  fputs (_("\
  \\\\      backslash\n\
  \\a      alert (BEL)\n\
  \\b      backspace\n\
  \\c      produce no further output\n\
  \\e      escape\n\
  \\f      form feed\n\
  \\n      new line\n\
  \\r      carriage return\n\
  \\t      horizontal tab\n\
```

```
  \\v      vertical tab\n\
"), stdout);
  fputs (_("\
  \\0NNN   byte with octal value NNN (1 to 3 digits)\n\
  \\xHH    byte with hexadecimal value HH (1 to 2 digits)\n\
"), stdout);
  printf (USAGE_BUILTIN_WARNING, PROGRAM_NAME);
  emit_ancillary_info (PROGRAM_NAME);
  exit (status);
}

/* Convert C from hexadecimal character to integer.  */
static int
hextobin (unsigned char c)
{
  switch (c)
    {
    default: return c - '0';
    case 'a': case 'A': return 10;
    case 'b': case 'B': return 11;
    case 'c': case 'C': return 12;
    case 'd': case 'D': return 13;
    case 'e': case 'E': return 14;
    case 'f': case 'F': return 15;
    }
}

/* Print the words in LIST to standard output.  If the first word is
   '-n', then don't print a trailing newline.  We also support the
   echo syntax from Version 9 unix systems. */

int
main (int argc, char **argv)
{
  bool display_return = true;
  bool posixly_correct = getenv ("POSIXLY_CORRECT");
  bool allow_options =
    (! posixly_correct
     || (! DEFAULT_ECHO_TO_XPG && 1 < argc && STREQ (argv[1], "-n")));

  /* System V machines already have a /bin/sh with a v9 behavior.
     Use the identical behavior for these machines so that the
     existing system shell scripts won't barf.  */
  bool do_v9 = DEFAULT_ECHO_TO_XPG;

  initialize_main (&argc, &argv);
  set_program_name (argv[0]);
  setlocale (LC_ALL, "");
  bindtextdomain (PACKAGE, LOCALEDIR);
  textdomain (PACKAGE);

  atexit (close_stdout);

  /* We directly parse options, rather than use parse_long_options, in
     order to avoid accepting abbreviations.  */
  if (allow_options && argc == 2)
    {
      if (STREQ (argv[1], "--help"))
        usage (EXIT_SUCCESS);
```

```
      if (STREQ (argv[1], "--version"))
        {
          version_etc (stdout, PROGRAM_NAME, PACKAGE_NAME, Version, AUTHORS,
                       (char *) NULL);
          return EXIT_SUCCESS;
        }
    }

  --argc;
  ++argv;

  if (allow_options)
    while (argc > 0 && *argv[0] == '-')
      {
        char const *temp = argv[0] + 1;
        size_t i;

        /* If it appears that we are handling options, then make sure that
           all of the options specified are actually valid.  Otherwise, the
           string should just be echoed.  */

        for (i = 0; temp[i]; i++)
          switch (temp[i])
            {
            case 'e': case 'E': case 'n':
              break;
            default:
              goto just_echo;
            }

        if (i == 0)
          goto just_echo;

        /* All of the options in TEMP are valid options to ECHO.
           Handle them. */
        while (*temp)
          switch (*temp++)
            {
            case 'e':
              do_v9 = true;
              break;

            case 'E':
              do_v9 = false;
              break;

            case 'n':
              display_return = false;
              break;
            }

        argc--;
        argv++;
      }

just_echo:

  if (do_v9 || posixly_correct)
    {
```

```
while (argc > 0)
  {
    char const *s = argv[0];
    unsigned char c;

    while ((c = *s++))
      {
        if (c == '\\' && *s)
          {
            switch (c = *s++)
              {
              case 'a': c = '\a'; break;
              case 'b': c = '\b'; break;
              case 'c': return EXIT_SUCCESS;
              case 'e': c = '\x1B'; break;
              case 'f': c = '\f'; break;
              case 'n': c = '\n'; break;
              case 'r': c = '\r'; break;
              case 't': c = '\t'; break;
              case 'v': c = '\v'; break;
              case 'x':
                {
                  unsigned char ch = *s;
                  if (! isxdigit (ch))
                    goto not_an_escape;
                  s++;
                  c = hextobin (ch);
                  ch = *s;
                  if (isxdigit (ch))
                    {
                      s++;
                      c = c * 16 + hextobin (ch);
                    }
                }
                break;
              case '0':
                c = 0;
                if (! ('0' <= *s && *s <= '7'))
                  break;
                c = *s++;
                FALLTHROUGH;
              case '1': case '2': case '3':
              case '4': case '5': case '6': case '7':
                c -= '0';
                if ('0' <= *s && *s <= '7')
                  c = c * 8 + (*s++ - '0');
                if ('0' <= *s && *s <= '7')
                  c = c * 8 + (*s++ - '0');
                break;
              case '\\': break;

              not_an_escape:
              default:  putchar ('\\'); break;
              }
          }
        putchar (c);
      }
    argc--;
    argv++;
```

```
          if (argc > 0)
            putchar (' ');
        }
    }
  else
    {
      while (argc > 0)
        {
          fputs (argv[0], stdout);
          argc--;
          argv++;
          if (argc > 0)
            putchar (' ');
        }
    }

  if (display_return)
    putchar ('\n');
  return EXIT_SUCCESS;
}
```

13. BusyBox (alternative to GNU on Linux)

```
int echo_main(int argc, char **argv)
{
        struct iovec io[argc];
        struct iovec *cur_io = io;
        char *arg;
        char *p;
#if !ENABLE_FEATURE_FANCY_ECHO
        enum {
                eflag = '\\',
                nflag = 1,  /* 1 -- print '\n' */
        };
        arg = *++argv;
        if (!arg)
                goto newline_ret;
#else
        char nflag = 1;
        char eflag = 0;

        while (1) {
                arg = *++argv;
                if (!arg)
                        goto newline_ret;
                if (*arg != '-')
                        break;

                /* If it appears that we are handling options, then make sure
                 * that all of the options specified are actually valid.
                 * Otherwise, the string should just be echoed.
                 */
                p = arg + 1;
                if (!*p)        /* A single '-', so echo it. */
                        goto just_echo;

                do {
                        if (!strchr("neE", *p))
                                goto just_echo;
                } while (*++p);

                /* All of the options in this arg are valid, so handle them. */
                p = arg + 1;
                do {
                        if (*p == 'n')
                                nflag = 0;
                        if (*p == 'e')
                                eflag = '\\';
                } while (*++p);
        }
 just_echo:
#endif

        while (1) {
                /* arg is already == *argv and isn't NULL */
                int c;

                cur_io->iov_base = p = arg;

                if (!eflag) {
```

```
                        /* optimization for very common case */
                        p += strlen(arg);
                } else while ((c = *arg++)) {
                        if (c == eflag) {
                                /* This is an "\x" sequence */

                                if (*arg == 'c') {
                                        /* "\c" means cancel newline and
                                         * ignore all subsequent chars. */
                                        cur_io->iov_len = p - (char*)cur_io->iov_base;
                                        cur_io++;
                                        goto ret;
                                }
                                /* Since SUSv3 mandates a first digit of 0, 4-digit octals
                                * of the form \0### are accepted. */
                                if (*arg == '0' && (unsigned char)(arg[1] - '0') < 8) {
                                        arg++;
                                }
                                /* bb_process_escape_sequence can handle nul correctly */
                                c = bb_process_escape_sequence( (void*) &arg);
                        }
                        *p++ = c;
                }

                arg = *++argv;
                if (arg)
                        *p++ = ' ';
                cur_io->iov_len = p - (char*)cur_io->iov_base;
                cur_io++;
                if (!arg)
                        break;
        }

 newline_ret:
        if (nflag) {
                cur_io->iov_base = (char*)"\n";
                cur_io->iov_len = 1;
                cur_io++;
        }
 ret:
        /* TODO: implement and use full_writev? */
        return writev(1, io, (cur_io - io)) >= 0;
}
#endif
```